Broadview®
www.broadview.com.cn

零基础学大数据算法

王宏志　林可　编著

中国工信出版集团
電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

作者简介

王宏志

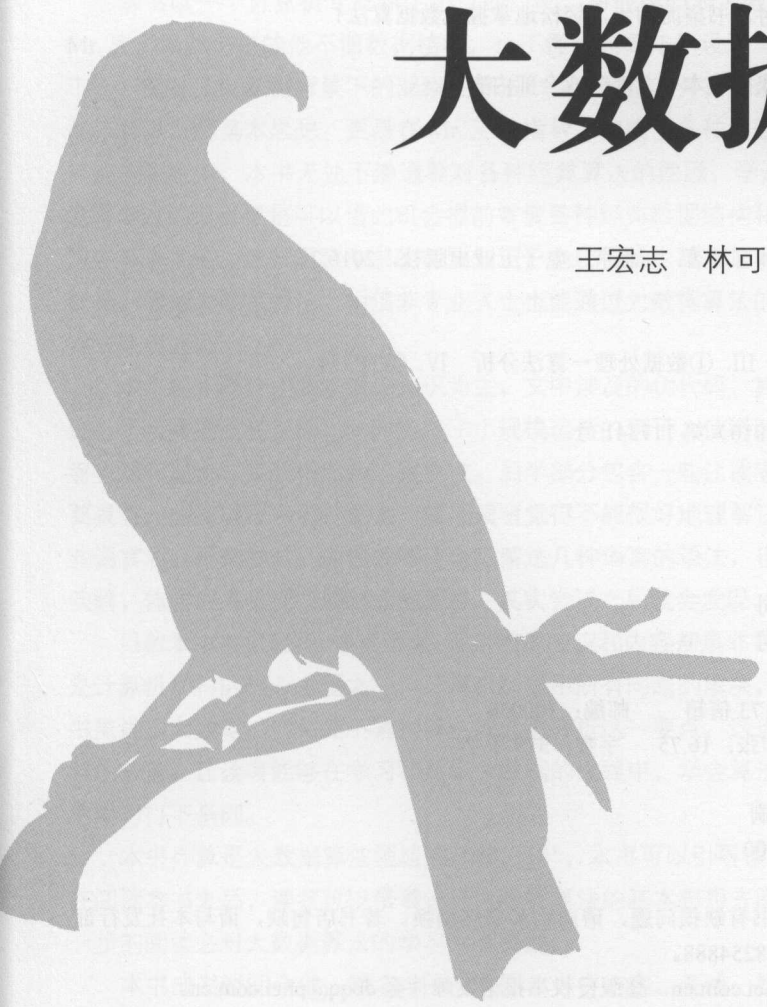
哈尔滨工业大学计算机科学与技术学院教授、博士生导师。其研究方向为大数据管理与分析、数据质量、图数据管理。发表学术论文170余篇，出版学术专著两本，出版国内首部《大数据算法》教材，其论文被SCI/EI检索110余次，他引500余次，其中7篇论文发表于顶级国际会议上。获得微软学者、IBM博士英才等称号，先后主持国家自然科学基金重点项目、国家支撑计划课题、国家博士后特别资助等10余个项目，还参加了国家973项目、863项目、自然科学基金重点项目等多个项目。他担任4个国际期刊的编委，现任CCF哈尔滨分部秘书长、ACM SIGMOD中国秘书长、中国计算机学会学术工作委员会委员、CCF高级会员、中国数据库专业委员会委员、中国大数据专家委员会通信委员、中国计算机应用专业委员会委员。

林可

现效力于哈尔滨工业大学计算机系海量数据研究中心，前支教教师。从事海量数据计算、分布式系统、感知系统等方面的研究，有着丰富的研究和项目经验，爱技术也爱生活的文艺极客范儿，也是一位在大数据新天地中展翅欲飞的后生。

零基础学大数据算法

王宏志　林可　编著



電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内 容 简 介

本书是通俗易懂的大数据算法教程。通篇采用师生对话的形式，旨在用通俗的语言、轻松的气氛，帮助读者理解大数据计算领域中的基础算法和思想。

本书由背景篇、理论篇、应用篇和实践篇四部分组成。背景篇介绍大数据、算法、大数据算法等基本概念和背景；理论篇介绍解决大数据问题的亚线性算法、磁盘算法、并行算法、众包算法的基本思想和理论知识；应用篇介绍与大数据问题息息相关的数据挖掘和推荐系统的相关知识；实践篇从实际应用出发，引导读者动手操作，帮助读者通过实际程序和实验验证磁盘算法、并行算法和众包算法。

在讲解每一个大数据问题之前，本书都会介绍大量的经典算法和基础数据结构知识，不仅可以帮助学习过数据结构与算法、算法设计与分析等课程的同学复习，同时能够让入门的“小菜鸟”们，不会因为没有学习过经典算法而对本书望而却步，轻松地掌握大数据算法！

图书在版编目（CIP）数据

零基础学大数据算法 / 王宏志，林可编著. —北京：电子工业出版社，2016.7
ISBN 978-7-121-28937-8

Ⅰ. ①零… Ⅱ. ①王… ②林… Ⅲ. ①数据处理－算法分析 Ⅳ. ①TP274

中国版本图书馆 CIP 数据核字（2016）第 117341 号

策划编辑：张月萍
责任编辑：葛 娜
印 刷：北京京师印务有限公司
装 订：北京京师印务有限公司
出版发行：电子工业出版社
北京市海淀区万寿路 173 信箱 邮编：100036
开 本：787×980 1/16 印张：16.75 字数：379 千字
版 次：2016 年 7 月第 1 版
印 次：2016 年 7 月第 1 次印刷
印 数：4000 册 定价：59.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线：（010）88258888。

前　言

这是一个互联网的时代，也是一个大数据的时代。经常有朋友问起：什么是大数据？大数据是做什么用的？我们为什么要研究大数据？应该怎么研究大数据？在寻找这些问题的答案时，许多朋友找到的内容常常是专业的概念、复杂的公式和难懂的“算法”，这让他们望而却步。很多计算机专业的新生或低年级学生在听到大数据的概念后对其非常好奇，却因没有足够扎实的专业基础知识而无法认识和理解大数据问题，更无法对大数据问题给出很好的解决办法。于是，笔者决定编写一本新生乃至非专业人士也能读懂的大数据算法教程。

本书以一个计算机专业新生小可的口吻，将他内心对大数据的好奇一一询问学识渊博的Mr. 王。虽然书中的他不懂数据结构，也不懂经典算法的设计与分析，却在 Mr. 王的耐心教导下一一突破了大数据背景下的亚线性算法、磁盘算法、并行算法、众包算法，了解了数据挖掘和推荐算法的基本思想，更是在 Mr. 王的指导下完成了各种大数据算法的实现。在所有大数据算法的讲授中，本书无处不渗透着对各种经典算法的回顾，学过的读者可以进行充分的复习，没有学过的读者更是可以借此机会提前掌握各种经典数据结构和经典算法，使得其在今后的学习中事半功倍。这些贯穿全书的前置知识，也使得新生甚至是非专业人士能够通过本书读懂大数据，读懂大数据算法。相信非专业人士也能通过大数据算法的思想，重新认识大数据，并获得一些启迪。

本书前半部分主要以理论知识为主，文中涉及的伪代码、算法等均以简单易懂的自然语言进行了步骤描述和解释，同时给出了小规模运行的例子，使得读者可以轻松地理解。同时，笔者也深知理论与实践相结合的重要性。后半部分包含一些让读者进行实践的实验和程序。这需要具有一些基础程序设计能力，如果读者觉得不能很好地理解它们也不要心急，可以待学会这些语言后，再来尝试。即使并不完全理解这几种语言的语法，也可以按照书中详尽的步骤进行实验，体会成果出现在屏幕上的喜悦，其实尝试之后就会发现，阅读它们并不困难。

虽然本书气氛轻松、语言活泼，但讲授的知识和内容却是非常“专业”的，“算法设计与分析”是计算机学科的核心主题之一，计算机科学中所有问题的解决，都离不开算法设计与分析。本书虽讲解大数据，但无处不紧扣算法设计与分析这一要点，这也让本书带上了浓厚的计算机学科的味道，让读者能够在学习和认识大数据的过程中，学会算法设计与分析，为其他领域知识的学习打下基础。

本书亦算是大数据算法领域的“敲门砖”，本书可以引导读者形成设计大数据算法的思维。在阅读本书之后，读者可以带着设计大数据算法的基本思想去阅读更加深入的专著或论文，进一步的阅读必对大数据算法的学习大有裨益。

本书成书时间仓促，笔者水平亦有限，书中内容、表述、推理等方面的各种不当之处在所难免，敬请各位读者在阅读过程中不吝提出宝贵意见。

目　录

第 1 篇　背景篇

第 2 篇　理论篇

第 3 篇　应用篇

第 4 篇　实践篇

咚咚咚。

一天下午，王老师的门被敲响了。

Mr. 王：请进。

门被轻轻地推开了，随后被有礼貌地关上了。

Mr. 王：你就是小可吧?

小可：是的，王老师您好，我就是前几天与您联系的那个学生，我想学习些大数据方面的知识。

Mr. 王：好啊，咱们可以一起讨论，看你不太面熟，你是计算机专业的学生吗?

小可：我是计算机专业的大一新生，会用程序设计语言完成一些很简单的程序设计，不过到目前为止我还没有学习过任何关于大数据算法的课程，我也并不了解什么是算法设计与分析，就连算法是什么都不太清楚，可是每天都能听见大家在讨论大数据的问题，我也很想了解大数据方面的内容，这样的基础我也能听懂大数据的内容吗？

Mr. 王：当然可以，我可以给你讲几节讨论课，相信学习过后，你就明白什么是大数据、如何分析大数据和应用大数据了。

小可：那真是谢谢王老师了。

Mr. 王：别客气，有问题直接来问我就可以了。

第 1 篇　背景篇

第1章 何谓大数据

1.1 身边的大数据

小可：王老师，那什么是大数据呢？

Mr. 王：你还真是一下就问了个很复杂的问题。其实大数据是一个很模糊的概念，很多学者和学术组织都对其提出过自己的定义，但是至今还没有公认的定义。我们先不谈确切的定义，先来举几个例子说明吧。你平常用社交网络吗？

小可：嗯，是的。

Mr. 王：你有很多好友吧？他们是不是每天都会发很多的状态和消息？

小可：是的，甚至有很多新闻我都是首先通过社交网络知道的。社交网络传递信息的速度真的很快，朋友们每天发布的状态我都看不完，而且不仅有原创的内容，还有很多来自他们好友的转载内容。

Mr. 王：其实社交网络上的这些信息就是一种典型的大数据。

小可惊讶地说：原来这就已经是大数据了？我一直以为大数据都在实验室里面呢。

Mr. 王：此言差矣，其实大数据就在我们身边。我们常用的社交网络上就有着非常巨大的信息量，虽然一个人发布的状态非常有限，但由于使用的人数众多，加之转载和评论，巨大的数据规模就使得社交网络信息无法在短时间内由人工或者由少量的几台计算机存储和管理。站在社交网络之外看待它，就会发现里面有很多且杂乱无章的信息和内容，同时其规模非常大。这就是大数据的一个典型例子。

小可恍然大悟地说道：哦，原来这就是大数据啊，那其实我每天都在接触大数据。

Mr. 王笑道：的确，大数据就在我们每个人的身边，随着信息时代的到来，我们每个人每天接触到的数据量都是非常大的。但你在查看这些消息的时候，有没有看到除字面内容以外的东西呢？

小可想了一下，说：好像没有什么，我关注的只是消息本身。

Mr. 王：我们研究大数据不只是能知道它的数据量很大，或者说仅仅研究如何把它们存储起来，我们还要发掘在大数据中隐藏的知识和有价值的信息。

小可：哦？大数据中隐藏着知识？

Mr. 王：是的，从表面上看，大数据可能只是一些简单的文本、杂乱的符号或者是一些数字的序列或者集合，但是从这些文本或者数字的背后，我们可以发掘其作为一个群体所具有的一些性质，从而发现一些对我们有意义、有价值的信息，所以我们才要研究大数据。

小可：大数据不是很大很大吗？那么我们研究它不就会变得很困难吗？

Mr. 王：不错，大数据的量很大很大，我们单单是把其中的信息逐个地访问一遍都很困难，所以发掘其中的知识就更加困难了，这就是研究大数据要解决的重要问题，也就需要我们这些研究大数据的人、热爱大数据的人加倍地努力了。

小可思考片刻后，说：那在超市里面，每年都会有很多人去买东西，他们的购物单上又会包含着很多内容，对超市来说，这些购物的记录就是“大数据”吧？而通过分析这些购物单，发现顾客更喜欢买哪些商品，这算不算一种通过大数据分析出的知识呢？

Mr. 王：很聪明嘛，你举了一个很好的例子。商业数据也是大数据的一个重要体现，超市购物的明细记录、公司运营的详细账目这些数据量都是很大的，处理起来非常费时费力，而其中又包含着有价值的信息，通过这些信息不仅可以分析出本年度公司的运营情况，同时可以指导下一年度公司的营销战略，这些数据对公司来说可谓是价值连城。

小可：那么大数据在别的方面又有哪些体现呢？

Mr. 王：你应该对生物遗传有所了解吧。

小可点点头道：是的，人体通过 DNA 携带遗传信息。

Mr. 王：在医疗和生物计算领域中，每次对 DNA 序列的分析都会产生大量的数据，这个数据量已经不是用 GB 可以衡量的了，甚至要达到 PB 级别或者更大。而这么大的数据，不仅计算机的内存装不下，而且一般计算机的硬盘都已经存不下了。即使是扫描一遍，在上面发现一个小序列都需要一些时间，在这些数据上面做分析将是一件更困难的事情。这也是一种大数据。

不仅在生物学中如此，而且在很多科学仪器的使用过程中也都会产生大量的数据，比如天文观测、显微观测，现在逐渐应用的传感器和传感器网络在使用过程中都会记录下大量的数据。这些仪器不停地记录下的数据，都涉及如何存储、如何分析研究的问题，这些都是大数据。

生活中的大数据

小可：嗯。

Mr. 王：那我们就给大数据下个定义吧。

定义 1：所涉及的数据量规模巨大到无法通过人工，在合理时间内达到截取、管理、处理并整理成为人类所能解读的信息。（Dan Kusnetzky, What is“Big Data”？）

定义 2：不用随机分析法（抽样调查）这样的捷径，而采用所有数据的方法。（维克托·迈尔 - 舍恩伯格、肯尼斯·库克耶，“大数据时代”）

定义 3：“大数据”是需要新处理模式才能具有更强的决策力、洞察发现力和流程优化能力的海量、高增长率和多样化的信息资产。（“大数据”（Big Data）研究机构 Gartner）

有了前面的那些例子，这些定义是不是相对好理解一些呢？

小可：嗯，我懂了。

1.2 大数据的特点和应用

Mr. 王：大数据具有较大的数据量，和一般的数据相比，其具有如下一些特点。

- 在数据量上，大数据是通过各种设备产生的海量数据，其数据规模极为庞大，远大于目前互联网上的信息流量，PB 级别将是大数据的常态。
- 在多样性上，大数据种类繁多，在编码方式、数据格式、应用特征等多个方面存在差异性，多信息源并发形成大量的异构数据。
- 在速度上，涉及感知、传输、决策、控制开放式循环的大数据，对数据实时处理有着极高的要求，通过传统数据库查询方式得到的“当前结果”很可能已经没有价值。
- 在价值上，数据持续到达，并且只有在特定时间和空间中才有意义。

Mr. 王：我们分析大数据、研究大数据，是希望能够利用它们获得我们需要的知识。我们可以利用大数据进行：

- 预测
- 推荐
- 商业情报分析
- 科学研究

等发现大数据中的价值，使用大数据、利用大数据的过程。由此可知，对大数据的研究还是非常重要而有意义的。

小可：有种大数据中有黄金的感觉啊。

Mr. 王：正是如此，从大数据中挖掘出来的价值，真是难以估量啊。今天时间不早了，你先回去吧，下节课咱们讨论一下关于算法的问题，要讨论大数据算法，必须先了解算法的相关知识。

小可：谢谢老师，那我下次再来。

第2章 何谓算法

2.1 算法的定义

小可：王老师您好，了解了什么是大数据，今天我来听听关于算法的内容。

Mr. 王：你好，想要学懂大数据算法，算法的基础知识是一定要了解的，你必须要知道如何设计和分析算法，我们才能谈及如何在大数据集合上研究算法。

小可：我经常会听到“算法”这个词，计算机专业的同学总会提到，那么到底什么是算法呢？

Mr. 王：至今为止，算法都没有一个准确的定义。每个计算机科学家和工程师都在设计算法、使用算法、分析算法、实现算法，但对于算法的定义依然是众说纷纭，很多书籍都曾经给出自己的定义，与其说那是一个定义，不如说都是对算法的一种诠释。的确，算法是一个很模糊、抽象的概念。

小可：那么我该如何搞懂什么是算法呢？

Mr. 王：在解释“算法”这个概念之前，我们首先来谈谈，一个计算机科学家是如何解决问题的。我先问问你，计算机是用来做什么的呢？

小可：计算、办公、游戏、影音娱乐，还有上网。

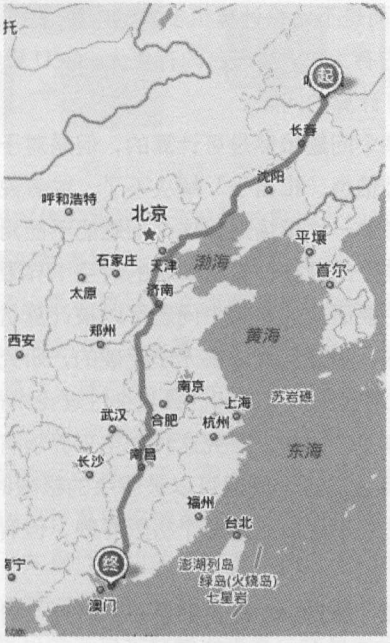
Mr. 王：不错，宽泛地谈起计算机的用途真是数不胜数。不过总结起来，其实计算机做的事情就一个——解决问题。

小可：解决问题？

Mr. 王：对，生活中有很多问题，其中有些问题人工解决起来很费时费力，于是我们发明了许多工具，在这个层面上，其实计算机也是一种工具，本质上它就是解决问题的一种工具。

比如：

- 升空卫星的轨道是怎样的?
- 从哈尔滨到深圳，走怎样的一条路线最短、最省路费?
- 模拟一次比赛的结果，分析到底谁的胜算更大?
- 我们希望知道，某个游戏或者博弈中是不是有必胜的策略?



小可：嗯，其中有些问题人工解决起来确实很费劲。那么计算机科学家又是如何解决这些问题的呢?

Mr. 王：首先，如果希望计算机能真正地解决一个实际问题，我们先要将现实世界中的事物转化为模型，这个模型可以被计算机理解和处理，它可以表示成数据和指令等。这个过程我们称之为**建立模型**。在此过程中，我们需要把一个实际问题抽象成计算机可以理解的语言，或者说计算机可以理解的问题，才可以用计算机求解。

小可：哦，这就是所谓的“建模”吧。

Mr. 王：其次，我们要知道这个问题是不是可计算的。计算机可以解决很多问题，也有很多问题解决不了。那么，由此诞生的，研究一个问题是不是计算机可计算的、可解的计算机科学分支叫作**可计算理论**。

小可：还有计算机解决不了的问题吗?

Mr. 王：当然，比如著名的“停机问题”。在这方面做出卓越贡献的科学家是非常著名的阿兰·图灵。图灵曾经提出过很多对计算机科学产生深远影响的理论，直到现在，我们使用的电

子计算机在模型上依然可以称之为“图灵机”。停机问题在很多资料中也称作“图灵停机问题”。图灵已经进行了证明，停机问题是不可计算的。

小可：那是不是说，我的计算机内存太小、CPU 太慢，有些特别大型的问题在我这里就“计算不出来”，这就是一个不可计算的问题呢？

Mr. 王：不，这是不对的，不可计算的问题并不是出于 CPU 速度和内存大小等资源的限制而无法在一定的时间内完成，而是不论给计算机多大的内存、给它多快的 CPU 都是无法求解的。如果你的计算机 CPU 太慢、内存太小，在一台内存更大、CPU 更快的机器上，还是能够求解的，那么这样的问题不是一个不可计算的问题。

不过这里有一个问题：某个问题虽然是可计算的，但是对于这个问题我们急着要的结果要算几年时间，那么这个问题恐怕也“相当于”解决不了，或者说交给计算机解决已经没有什么意义了。所以我们必须要知道的一件事情是，这个问题能不能用计算机在我们可以接受的时间或者空间界限内解决。研究某个问题被计算机解决的时空下界限的计算机科学分支称为**计算复杂性理论**。计算复杂性理论主要研究的是某个问题可以被计算机求解的时间和空间下界限。它研究给出的问题的时空下界限，是任何算法都无法突破的，研究比下界限更快的算法是没有意义的，当发现某一个算法已经足够快到可以和计算复杂性理论中得出的下界限是同一个级别时，我们就不必再去提升它的效率了。同时，研究这种时空下界限有另一方面的目的，就是可以用来评价我们现在设计出来的算法与这个问题可以被解决的极限时间空间界限还有多远的距离，很多时候我们设计出来的算法还不足以达到这个极限界限，但有了这个界限，可以给我们接下来的研究指明一个方向。

总结起来，可计算理论和计算复杂性理论都是一个研究“问题”的范畴。

研究过问题之后，我们要考虑的就是如何去解决问题。这也是计算机作为一种工具的重要属性。想要解决问题，就需要我们设计算法。

小可：那么到底什么是算法呢？

Mr. 王：这里我们还是举个生活化的例子吧。比如，我们现在想要煮一锅汤，这就是一个问题。根据生活经验，我们认为它是可解的（可计算分析），也是理论上可以在我们接受的时间范围内解决的（计算复杂性分析）。这时，需要我们设计一个解决它的方法或者说一系列步骤。

小可：我想想，煮汤我们可以想到的步骤就是洗菜、切菜、烧水、煮汤、出锅。哈哈。

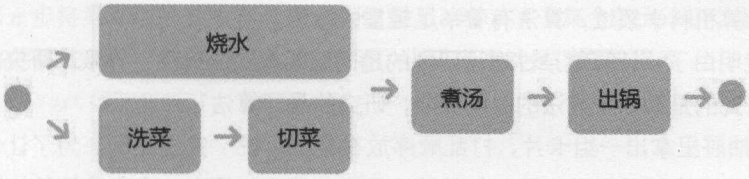
煮汤的“算法”

Mr. 王：很好，你已经在设计一个算法了。

小可吃惊地说：啊？！这就是一个算法了？

Mr. 王：从广义上讲，这种解决一个问题的一系列准确完整的步骤，就是算法。

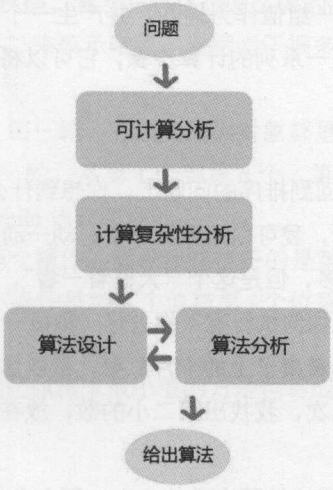
小可：哦，原来这已经是一个算法了？等等，王老师，我觉得我设计的这个步骤有些缺陷，如果按照这个步骤，在烧水的过程中，人一直是闲着的。如果先把水烧上，再去洗菜、切菜，是不是能更快地把汤煮完呢？也就是说，一个更好的步骤是，烧水、洗菜、切菜、煮汤、出锅。



煮汤的“算法”改进版

Mr. 王：不错嘛，你刚刚很好地分析了自己设计的算法。你注意到了一个算法具有的缺陷，并且分析它、改进它，提出了更好的新算法。但是计算机科学中的算法分析要比这个略复杂些，一会儿我会给你讲解，为了理解大数据算法，必须要了解如何分析算法。

在计算机科学中，研究算法的设计和评价算法“好坏”的分支，称为**算法设计与分析理论**。它研究如何去设计解决问题的算法，同时给出一个对算法在计算机中执行的时间和空间效率，评价这个算法是不是足够快、占用的空间足够小。到目前为止，高速的 CPU 和高速大容量的寄存器、缓存和内存依然是很昂贵的计算资源。另外，CPU 的运算速度和内存容量相对目前的大数据来说依然是不够的。所以设计高效率的算法，一方面是为了节约时间；另一方面也是为了节省金钱。从另一个角度讲，如果计算机的速度非常快、内存非常大，而且程度可以逼近无穷的话，恐怕我们就不再需要对算法进行分析和改进了，只要结果正确就可以了。不过从目前的数据量、研究的问题以及计算机的运算存储能力来看，还是有很大差距的。



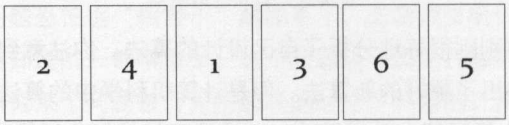
计算机科学解决问题的办法

小可：嗯，不仅仅是设计算法，看来分析算法也是非常有必要的，设计一个好的算法可以事半功倍。

Mr. 王：算法设计与分析这一部分也是计算机科学解决问题最重要的部分，是计算机科学的核心所在。在科学的发展史上，也正是因为算法的出现，才使得计算机科学得以独立成为一门学科。在计算机科学领域，算法有着举足轻重的地位。

小可：我明白了，计算复杂性理论研究的是问题能解决的时空下界限，研究的是“问题”，而算法分析研究的是某一个方法的时间界限，研究的是“算法”。

Mr. 王从抽屉里拿出一组卡片，打乱顺序放在桌上，说：完全正确。为了让你了解计算机中的算法，我们举个机器能够实际运行的算法例子。我在这里举一个比较简单的问题。



排序

比如现在有一组数字，我们希望将它们从小到大排序。这是算法设计中一类很基础也是很重要的问题，叫作“排序”。当我们要设计一个算法时，首先要分析它的输入输出。如果将算法视作一个机器的话，我们要将所需要处理的数据当作“原料”放进机器，然后经过机器处理将“成品”从机器中取出来。放进机器里的“原料”就是算法的输入，而取出来的“成品”就是输出。

在这个问题中输入输出应该是什么呢？

小可：输入是一组数字，输出是由这组数字构成的从小到大的序列。

Mr. 王：嗯。算法取一个或一组值作为输入，并产生一个或一组值作为输出，算法是一个定义良好的计算过程，或者说是一系列的计算步骤，它可以将输入数据转化为输出结果。这就是算法在计算机中的定义。

小可：原来这就是算法啊。

Mr. 王：没错。好了，咱们回到排序的问题上，你想到什么方法了吗？

小可：这应该有很多方法吧，我可以大概看一看，动一动就可以排出来了。

Mr. 王：排序的方法确有很多，但是这个“大概看一看”，计算机可做不到，这违背了算法的确定性，定义不明确，计算机如何执行呢？你去想一种每一步都可以确定执行的方法。

小可一边思索着，一边摆弄着桌子上的卡片：嗯，可以这样，第一次，我找出整个集合里面最小的数，放在第一位；第二次，我找出第二小的数，放在第二位。依此类推，直到所有的数都被放进序列中。

Mr. 王：假设计算机每次只能比较两个数的大小，那么应如何发现一组数里的最小值呢？

小可想了想，看着桌子上的卡片：这的确是个问题……可以这样做，我们先假设第一个数就是最小值，然后用后面的数与之比较，一旦发现有比它小的数，那么这个数就可以作为新的假设最小值，直到扫描了整个序列。

Mr. 王：不错，这样算法的步骤就被有效地具体化了。我们每一轮都执行选取最小值这个工作，这样第 n 步将第 n 小的数放在了第 n 个位置上，当 n 等于集合的大小时，就成功排列了。这种实现排序的算法叫作“选择排序”。我们用伪代码描述就是下面的程序：

```
Selection-Sort(A,n)
begin
    i ← 1;
    while i<n
    do
        min_position ← Find the minimal from A[i] to A[n];
        swap (A,i, min_position);
        i ← i + 1;
    end while
end
```

小可：伪代码是什么？

Mr. 王：伪代码是用来描述算法的一种语言，不是真正的计算机程序，但算法被描述成伪代码后，程序员就可以很容易地翻译成任何一种计算机语言了，这一步骤叫作算法的**实现**。而在算法被计算机语言实现后，可以形成相应的**软件系统**。我们今后的讨论也将常常用到伪代码来描述算法。伪代码并没有唯一的标准，很多书籍上使用的伪代码都是不同的，有时甚至就是英语或者中文句子。但是多数伪代码都是非常容易读懂的，具有一点编程基础的人都能够将它们实现为真正的计算机程序。

Mr. 王：我们来读读这段伪代码，看看它具体是怎么做的。

在伪代码中，我们常用“←”来表示赋值，它相当于很多高级语言中的等号“=”，它的意思就是把右边的值赋给左边。

就像我们之前描述的那样，每一轮，我们处理的对象都是还没有被排序的部分，在伪代码中体现的就是不断增加的 i。第一轮，从第 1 个到第 n 个；第二轮，从第 2 个到第 n 个。这是由在大多数高级语言中都有的 while 语句实现的。

在每一轮中，我们都进行两个操作。假设现在执行的是第 i 轮，第一个操作是从未排序的部分中选出一个最小值；第二个操作是将这个值与第 i 个位置进行交换，也就做到了第 i 轮将第 i 小的数放到第 i 个位置上。

如果希望具体一点的话，则可以将求最小值的方法也写成伪代码：

```
findMin(A,start,end)
begin
    i ← start
    min_pos ← i
```

```
    while i <= end
    do
        if A[i] < A[min_pos] then
        min_pos ← i
        end if
    end while
    return min_pos
end
```

我们首先假设第一个值是最小值，然后扫描整个数组，一旦发现有比它还小的值，那么这个值就假设为最小值。这里始终更新的不是最小值，而是最小值的所在位置，然后通过这个位置来访问最小值。如果访问最小值的函数希望返回最小值的话，那么只需要稍作修改即可，这个就留给你回去修改了。

需要注意的一点是，我这里使用的伪代码中的数组下标是从 1 开始的。而像 C 语言这样的很多高级语言都是从 0 开始的，不过相信聪明的你一定能够在实现它的时候注意到这个问题并进行相应的调整。

swap 这个函数非常简单，相信有一点编程基础的人都可以实现。你一定没问题吧?

小可：没问题，我觉得像这样写就可以了：

```
swap (A,i,j)
begin
    temp ← A[i]
    A[i] ← A [j]
    A[j] ← temp
end
```

Mr. 王：至此，我们就完成了一个非常简单的算法——选择排序的设计。

小可：哦，那么我设计的这个算法怎么样呢?

Mr. 王：其实，这并不是一个很好的算法，它的时间复杂度是 $O(n^2)$，而计算复杂性理论已经成功地证明了，基于比较的排序方法，最低的时间复杂度是 $O(n\log n)$。

小可：听不懂了。

Mr. 王：嗯，下一部分课我们就来讲讲具体如何评价一个算法。不论是大数据算法还是经典算法，算法的分析都是非常重要的，评价一个算法有助于考虑是否在某个问题的求解、工程的实现、系统的设计上使用该算法，同时也非常有助于通过改进而派生出新的算法。

2.2 算法的分析

小可：嗯，我觉得评价一个算法的最基本方式就是看它运行得快不快。

Mr. 王：嗯，这是重要的考量标准之一。研究算法运行得快不快的指标叫作**时间复杂度**。

而在评价算法的同时还要考察其对内存空间的占用情况，也就是**空间复杂度**。这两个指标对于评价算法来说是最重要的。

小可：时间和空间都是计算机的资源，好的算法应该较少地占用资源而得出结果。

Mr. 王：对。我们先从时间复杂度开始探讨，空间复杂度与之类似，只不过是面向内存空间的。

小可：那时间复杂度是不是就是指一个算法运行的时间长短呢？

Mr. 王：不，这是一个常见的误解，算法的时间复杂度并不是指一个算法实际运行的时间。举个简单的例子，要访问一个集合中的每个数据，这在计算机科学中称为**遍历**。可以考虑一下，对一个只有 10 个数据的集合，和对有 10000 个同类型数据的集合使用同一种算法进行遍历，时间显然是不一样的。一个坏的算法在小的数据集合上，很可能比一个好的算法在大的数据集合上运行得要快。另外，现在计算机的发展速度很快，将同样的算法和同样的数据在不同的计算机上运行，得出结果的速度往往也不同。所以从这个角度来看，只用得出结果的时间这一指标来衡量，是不够准确和恰当的。

小可：对，算法的执行时间还跟数据量大小有关，也跟所使用的计算系统有关。

Mr. 王：的确，所以我们必须要考虑输入的**数据规模**。在算法分析中，这个数据规模常常用 n 来表示。对于不同的问题，n 的单位也是不同的。在研究刚才的排序时，n 代表的是有 n 个数；在研究数据库的查询时，n 就表示有 n 条记录等。我们可以将关于数量级 n 的运行时间复杂度记为 $T(n)$。

我们就用选择排序法来做个例子吧。这个算法可以分为两个部分：第一，我们要找出在未排序部分里的最小值并放在该放的位置上；第二，持续执行第一部分，直到所有的数据都已经排好序。假设进行比较需要消耗一个常数 c 的时间，中间更新假设最小值的时间和进行交换的时间我们暂时不考虑，那么进行一次扫描需要多久？

小可：那就是 n 和 c 的乘积 cn。

Mr. 王：很好，但是每一次进行选择排序时，集合的大小都会缩小 1，假设每次都发生了交换，你说说看，整个算法的复杂性可以如何衡量？

小可：这样的扫描需要进行 $n-1$ 次，因为只剩下一个数时就不用比了。而每次进行比较的元素次数分别是 $1, 2, 3, \cdots, n-1$。这是一个等差数列。结合前面的式子，可以求得为 $T(n)=(n-1+1)\times(n-1)/2\times c=cn(n-1)/2$。

Mr. 王：于是，这个算法的复杂度为 $T(n)=O(n^2)$。

小可：这要怎么解释呢？

Mr. 王：假设 n 是一个很大的数，比如 10^{10}。那么常数 c 和 n 相比如何呢？

小可：常数 c 已经小到可以忽略不计了。

Mr. 王：同理，n 和 n^2 相比呢？

小可：那 n 也可以小到忽略不计了。

Mr. 王：在进行时间复杂度分析时，我们只保留多项式中的最高阶项。因为相比最高阶项而言，低阶项可以被忽略。同时，忽略其中的所有常数项系数。因为算法复杂度分析关注的是其数量级，而非具体的数值，当 n 是一个非常大的数时，n 的幂值和 c 相比已经非常悬殊了，c 的值小到对 n 的幂值不会产生太大的影响。另外，在计算复杂性理论中，有一个与这个问题相关的图灵机线性加速定理，这个定理证明了 cn^a 和 n^a 在复杂度分析上没有区别，感兴趣的话可以去查阅一下相关的资料。不管从哪个角度来看，都忽略在时间复杂度中所有的常数项系数。

回到我们的例子中，我们估计选择排序的运行时间是 $T(n)=cn(n-1)/2$，转化成多项式的形式就是 $T(n)=\frac{c}{2}\cdot n^2+\frac{c}{2}\cdot n$。根据前面的约定，忽略多项式中的低阶项，只保留最高阶项，就是 $\frac{c}{2}\cdot n^2$；还要忽略常数项系数，就是 n^2，所以 $T(n)$ 的数量级就是 $O(n^2)$。这里如果考虑对数组中元素进行交换时间的话，不难发现，当元素是顺序时交换次数最少，为 0 次；当元素是逆序时交换次数最多，为 $n-1$ 次；平均情况是一个关于 n 的线性函数，是 n^2 的低阶项，可以被忽略，没有影响到我们的分析结果。

小可：那么前面的大 O 表示什么呢？

Mr. 王：嗯，这里需要说明一下。很多时候当 n 不够大时，时间多项式中的低阶部分确实没有高阶部分大。比如对于常数较大的 n^2+c，当 n 比较小的时候，c 可能会比 n^2 还大，这就不符合 c 和关于 n 高阶项相比小到可以忽略这个要求。而我们在研究算法的复杂性时，并不关心 n 比较小的时候算法的运行时间，因为这时一些处理时间为常数项或者低阶项的操作对算法运行时间的影响是非常大的，此时的运行时间不足以真正地代表一个算法的性能。所以引入了一系列记号对复杂度进行定义，要对 n“较大”这个条件进行限制。

$O(g(n))$ 表示这样的一系列函数 $f(n)$，存在正常数 c 和 n_0，对于所有的 n 大于 n_0，有 $0\leqslant f(n)\leqslant c(g(n))$。大 O 记号的含义是 $f(n)$ 比 $g(n)$ 低阶。换句话说，$g(n)$ 表示的是 $f(n)$ 的上界。

n_0 的存在保障了我们研究的范围是 n 足够大时，它使得高阶项可以充分地大于低阶项。

小可：原来 $T(n)=O(g(n))$ 就表示当 n 足够大时，$T(n)$ 要比 $g(n)$ 小啊。

Mr. 王：通俗地讲是这样的。如果一个算法的运行时间 $T(n)=O(g(n))$ 的话，则说明其运行时间的渐进上界是 $g(n)$。也就是说，对于足够大的输入规模 n，这个算法的运行时间不会超过 $g(n)$。举个例子，选择排序的时间复杂度为 $O(n^2)$，如果输入规模是 10^{10}，那么它的运行时间应该在 10^{20} 这个数量级。

小可：这样看来，选择排序的运行时间应该会很久吧？

Mr. 王：是的，$O(n^2)$ 这个数量级实际是比较大的，这说明选择排序并不是一个很好的排序算法。

另外还要注意一点，我们在实际做算法分析时，如果使用大 O 记号来表示一个算法的复杂度，所确定的 $g(n)$ 一定要足够紧确，这样才能最好地代表一个算法的复杂性如何；因为我们去找一个很大很大的 $g(n)$ 也能满足 $T(n)=O(g(n))$，不过这样对于分析这个算法来说，就没什么

意义了。

小可：比如一个算法的运行时间是 $T(n)=2n+3$，那么它就是一个 $O(n)$ 的算法，就定义而言，$T(n)=O(n^2)$ 也是成立的。但是后者就不够紧确，这种不够紧确的时间界限不能真正地反映一个算法运行的时间界限。

Mr. 王：不错，这种 $T(n)=O(n)$ 的算法，也叫**线性算法**，说明它可以在与输入同规模的时间界限之内得出结论，与输入规模呈线性关系。如果一个算法的复杂度比线性算法还低，它就可以称为**亚线性算法**。比如 $O(\log n)$、$O(\log\log n)$ 以及 $O(1)$。$O(1)$ 也可以称作常数时间算法，我们注意到 $\log n$ 可以不写对数的底数，这同样也是对常数的一种忽略。这样忽略的原因其实非常简单，因为有对数换底公式，比如 $10\log_2 n=2\log_{10} n$，在复杂度分析时，忽略前面的常数项系数，所以 $O(\log_2 n)$ 与 $O(\log_{10} n)$ 同阶。在常用的写法中，我们会忽略对数的底数，在复杂度中的 log 与 ln 都是同阶的。另外，如果某个算法的时间复杂度 $T(n)$ 可以表示为一个多项式，那么这个算法可以叫作**多项式算法**。对于一个不太大的数据规模，或者说在经典的计算理论中，我们认为这是一个易解问题；如果找不到多项式算法，比如找到的算法都是 $O(2^n)$ 级别的，我们会认为这是一个难解问题。

小可：嗯，当 n 比较大时，2 的 n 次幂比关于 n 的多项式要大得太多了。如果一个 $O(2^n)$ 的算法真的有 10^{10} 这样的输入规模，那可真是天文数字了。可是现实中真的有这样的算法吗？

Mr. 王：其实这种需要 $O(2^n)$ 的算法，还真的广泛存在。比如我要枚举一个集合的所有子集，你看看它的复杂度有多大。

小可：如果一个集合有 n 个元素，那么它的子集就有 2^n 个！

Mr. 王：于是，子集的枚举就非常容易产生 $O(2^n)$ 这种高阶的复杂度。相比之下，比较高阶的复杂度还有 $O(n!)$，指数函数和阶乘的增长速度都是非常快的，不难看出，在一个比较小的 n 值下，指数函数和阶乘都可以达到一个非常高的数量级。也就是说，一个拥有如此高阶复杂度的算法在同样的数据规模下运行时间往往是非常长的，一般来讲，这样的算法不是好的算法。就目前的知识看来，如果一个问题需要超过多项式时间界限的算法来解决，我们一般认为这个问题是一个难解问题。

小可：那计算机科学有没有对易解和难解问题进行一个相对严格的界定呢？

Mr. 王：有的，这里既然提到了多项式算法和易解难解问题，那么我们就简单来谈一谈 NP 完全性的问题，这有助于对后面一些问题的理解。真正的 NP 完全性讨论和复杂度归约是比较复杂的主题，一般要到硕士生阶段才会接触，这里我们只简单谈谈。

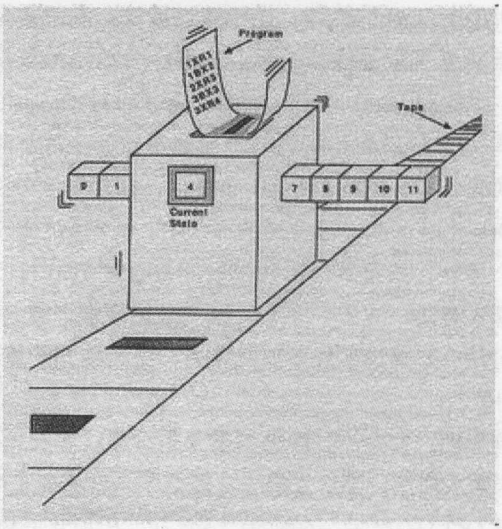
提到 NP 完全性，我们先要了解前面提到过的“图灵机”。

小可：我也很好奇，这个“图灵机”究竟是什么呢？

Mr. 王：想要理解图灵机，需要具有一定的自动机理论基础，最好先了解一下有穷自动机和下推自动机。这里我力图用浅显易懂的语言来解释它，所涉及的一些数学定义和形式化定义

就暂且不提了。

简单来说，图灵机是由一个读写头和一条两端无限延伸的纸带构成的。当然，也可以是多个读写头和多条纸带，科学家们已经证明，单带图灵机和多带图灵机是等价的。这里我们先谈谈单带图灵机。



图灵机图片

其中读写头可以在纸带上左右移动，可以在纸带上写下符号，也可以将纸带上的符号擦去（或者说写下“空”这个符号），还可以读取纸带上的符号。在读写头的内部，有一些“状态”，在读取不同的符号或者移动时，读写头的内部可以进行状态转换，在下一个阶段时，读写头可以根据状态和读取到的符号决定如何移动、读、写。

小可：这还是有一点抽象。

Mr. 王：这样吧，我们来设计一个图灵机，来看看图灵机是如何工作的。

比如，想让图灵机解决 2+3 这个算术题，我们就去编一个加法计算器的图灵机程序。

对于图灵机来说，它的一切都是可以定义的。为了方便起见，我们不使用十进制或者二进制形式，而是用纸带上的 1 个“1”表示数字 1，用 2 个“1”表示数字 2，依此类推。

小可：这也就是说，只要在纸带上画几个“1”就可以表示几了。

Mr. 王：另外，我们还要用一个符号来表示加号。这里其实选用任何符号都可以，但在计算机中常常使用二进制形式，那么除数字 1 以外就会用到数字 0。为了方便起见，我们用数字 0 来表示加号。那么这个算式可以表示成什么？

小可：我们要计算的算式是 2+3，那就是“110111”。

Mr. 王：很好，我们将它写在纸带上。为了让纸带上的空白区域更加方便地被图灵机识别，

我们用字母 B 表示空白（blank）。

小可：由于纸带是无限长的，那么纸带上留下的就是……BBBB110111BBBB……

Mr. 王：光有纸带还是不够的，还要对图灵机进行编程。我们先不去看图灵机程序，而是想一想，这应该怎么做？

小可：如果“11”表示 2，“111”表示 3，那么 2+3 的结果 5 就是“11111”。

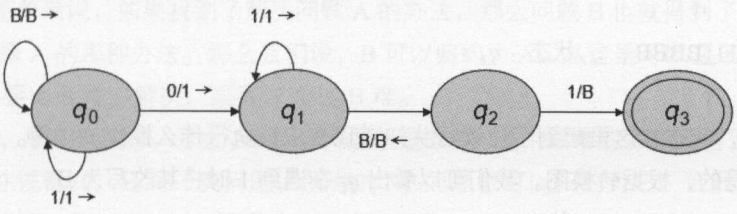
Mr. 王：很好，这个思路是对的，我们需要做的其实就是将纸带上的 110111 变成 11111。当然，一个加法计算器不能只对 2+3 可用，它也应该可以把 1011 变成 111，表示 1+2=3；可以把 11101111 变成 1111111，表示 3+4=7。

小可：其实这好像也不难，只要把中间的 0 去掉，然后把右边所有的 1 都往左挪就可以了。

Mr. 王：很好，其实图灵机就是这样做的。不过还可以做一个小小的改进，让图灵机执行起来更加方便，当图灵机遇到 0 时，我们把它变成 1，然后再把最后一个 1 抹掉，是不是也可以啊？

小可：嗯，这样和前面那种效果是一样的。

Mr. 王：好，我们用状态图来表示一下这个图灵机程序。



状态转换图

小可：这个好复杂啊。

Mr. 王：其实图灵机程序本质上就是读写头遇到了一个什么符号、如何转换状态、如何修改纸带上的符号、什么时候停机这样的一个组合，箭头连接符号用来表示状态的转换。上面的 X/Y →表示读到符号 X 时，将其改写为符号 Y，并且读写头右移（相应的“←”就表示左移）。

我们来看看它的执行结果。

刚开始，图灵机读到 B，根据转换图读写头向右移动：

BBB<u>B</u>110111BBBB　　状态：q_0

↑

遇到 1，读写头右移，又遇到 1，读写头继续右移：

BBBB1<u>1</u>0111BBBB　　状态：q_0

↑

读写头遇到 0 时，根据我们的设计，要将它改为 1，然后右移：

BBBB11<u>0</u>111BBBB　　状态：q_0

↑

同时，状态变为 q_1：

BBBB11<u>1</u>111BBBB　　状态：q_0->q_1

↑

右面的 1 和左面的一样，继续让读写头右移跳过它们即可，然后继续右移：

BBBB111111<u>1</u>BBBB　　状态：q_0

↑

继续右移，遇到了 B，这时我们知道右边已经没有 1 了，但是前面还有一个 1 我们没有将其抹掉，也就是此时纸带上留下的计算结果比正确的结果要多 1，根据程序继续执行，将其去掉。注意，此时状态要发生转换。想想看，如果没有一种状态记录读写头已经经过了扫描全部的 1 那个步骤的话，就会出现读写头返回去抹掉多余的 1 时遇到了 1，根据状态 q_1 上面的转换，又继续往右走，然后遇到 B 又返回的死循环情况。

BBBB111111<u>B</u>BBB　　状态：q_1->q_2

↑

BBBB111111<u>1</u>BBBB　　状态：q_2

↑

小可：哦，状态在这里起到了让读写头知道现在应该执行什么操作的作用。

Mr. 王：是的，根据转换图，我们可以看出 q_2 在遇到 1 时将其改写为 B。

BBBB11111<u>B</u>BBBB　　状态：q_2->q_3

↑

图灵机停机：

BBBB11111<u>B</u>BBBB　　状态：q_3

↑

小可：结果是对的，纸带上留下了 5 个 1！

Mr. 王：还可以多用几个算式执行一下这个图灵机，来验证我们设计的程序还是不错的。这是一个很简单的图灵机例子，不过可以很有效地说明图灵机是如何定义和工作的。

小可：嗯，我懂了。可是您前面说现在的计算机在模型上都可以称作图灵机，这个要如何理解呢？

Mr. 王：你能思考这个问题是非常好的。其实现在电子计算机可以解决的所有问题，都可以用图灵机解决，就用 2+3 这个例子，我们一开始将“算式”写在纸带上，相当于“输入”；图灵机的执行过程相当于计算机对问题进行处理；留在纸带上的结果相当于“输出”；状态转换图，相当于计算机程序；纸带在执行过程中相当于内存，读写头一部分是 CPU，同时也是读

写内存的设备。

小可恍然大悟，说：这么一说，好像确实和计算机挺像的。

Mr. 王：好，既然你初步理解了什么是图灵机，我们就来说说什么是易解问题和难解问题。

前面我们提到过多项式时间。如果一个问题可以用确定状态图灵机（DTM）在多项式时间界限内解决的话，我们称之为 P 问题；如果一个问题可以用不确定状态图灵机（NTM）在多项式时间界限内解决的话，我们称之为 NP 问题。

在这里，所谓的确定状态就是说如果在每一种状态下接收到一个特定的输入，它都会执行固定的状态转换和动作，这就是确定状态图灵机；反之，如果在某一种或多种状态下，对于某一个输入，它可能产生多种不同的状态转换，这就是不确定状态图灵机。而计算机只能表达确定状态图灵机，无法表达不确定状态图灵机，所以我们用计算机去解决 NP 问题的时间界限往往是指数的。

有这样一类问题，首先它是 NP 问题，其次所有的 NP 问题都可以归约为它，我们称之为 NP 完全问题（NP-complete）。

小可：什么是归约呢?

Mr. 王：简单来说，如果找到了解决问题 A 的办法，那么问题 B 也就得到了解决，而且正可以用解决问题 A 的那种办法。那么我们说，B 可以归约为 A。从这里可以看出，B 可以归约为 A，说明 A 要比 B 难以解决，或者说 A 比 B 难。

小可：哦，原来是这样，那么 NP 完全问题就是 NP 问题中最难的那些问题了?

Mr. 王：你说得对。还有一类问题，所有的 NP 问题都可以归约为它，但我们还无法判定它是不是 NP 问题。我们将这类问题称为 NP 难问题（NP-hard）。

小可：也就是说，我们还确定不了不确定状态图灵机能不能在多项式时间界限内解决它，那说明它的难度有可能比 NP 完全问题更高吧。那是不是说，如果我们恰好证明了一个 NP 难问题是 NP 问题的话，那么它就是 NP 完全问题了?

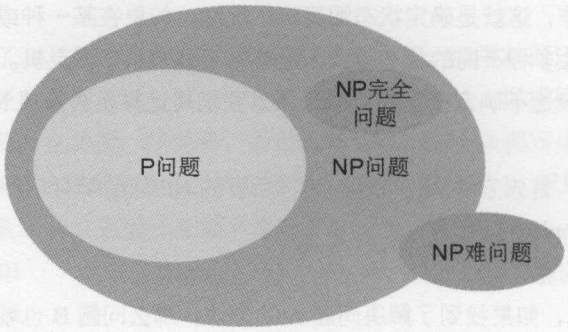
Mr. 王：是的，这说明你对 NP 问题的定义已经有了很好的了解。了解了这几类问题，我们不难发现有 $P \subseteq NP$，也就是说，P 问题都是 NP 问题。这是因为我们可以把确定状态图灵机看作不确定状态图灵机的一种特殊形式，只不过它的所有转换都是确定的，也就是说，所有的转换概率都是 1。

小可：嗯，的确是这样，P 问题都是 NP 问题。

Mr. 王：现在困扰计算机科学界最大的问题就是 P 是否等于 NP，这个问题被提出了多年，影响着很多问题的研究，至今未能得到解决。显然有 P 包含于 NP，但到目前为止 NP 是不是包含于 P 还不能确定。如果这个问题得到了证明，就说明 P=NP。在计算机科学界里，大量的疑团都指向 P 是否等于 NP。如果 P=NP，那么现在很多科学研究的结论将会被改写，很多新的结论也会被提出。但是证明这个问题是非常非常难的，当然，也有可能最后证明的结果是 P

不等于 NP。要证明 P=NP，就需要计算机科学界的学者们多多努力了。

好了，现在我们来给问题的难度排个序。首先有 P 包含于 NP，所以在难度上 P ⩽ NP。由于 NP 完全问题是 NP 问题中最难解决的，故 NP 完全问题会难于一般的 NP 问题，所以有 P ⩽ NP ⩽ NPcomplete。由于 NP-hard 和 NP-complete 同属的所有 NP 类都可以归约为它们的这种问题，而 NP-hard 还不能确定是不是 NP 问题，所以它应该更难一些，所以有 P ⩽ NP ⩽ NPcomplete ⩽ NPhard。我们一般认为 P 问题是易解问题，而 NP-complete 以上的就是难解问题。



P-NP 问题的关系

小可：嗯，我懂了。

Mr. 王：不过进入了大数据时代以后，易解和难解问题又相应地发生了一些变化，当数据规模并没有那么大的时候，多项式算法在求解很多问题时，算法的实际运行时间或许我们还可以接受，我们认为多项式算法还算是好的算法，能用多项式算法解决的问题还算是易解问题；但当数据量真的大到可以称之为“大数据”的时候，多项式算法的实际运行时间也会变得非常长，变得我们难以接受，这样多项式算法就已经不能满足我们对于很多大数据规模的问题求解。有时一个问题虽然是 P 问题，但是由于数据规模太大，也变得很难以解决，甚至当输入规模特别大的时候，在很多情况下就连线性算法也难以满足需求了。有些时候，我们就不得不去设计一些后面要讲的亚线性算法来匹配这些非常大的数据集合，以满足我们对它的速度要求。

小可：那有没有更快的算法？比如其运行时间与输入的数量级完全无关，如就是个常数项 c 呢？也就是说，这个算法不论输入 n 多大，它的运行时间都是一个特定的常数 t 呢？

Mr. 王：这样的时间界限记为 O(1)，我们称之为常数时间算法，这样的算法一般来说是最快的，因为它与输入规模完全无关，不论输入规模 n 多么大，我们都可以用一个与输入规模 n 无关的常数时间得出结论，相比于巨大的 n 来说，这个常数在数量级上已经微乎其微了。

小可：哦，这也体现了只关心数量级，而不关心具体数值的思想。

Mr. 王：另外，与大 O 记号类似，常用的记号还有 Θ，$\Theta(g(n))$ 表示函数 $f(n)$ 构成的集合，存在 n_0, c_1, c_2，当 $n \geqslant n_0$ 时，$0 \leqslant c_1 g(n) \leqslant f(n) \leqslant c_2 g(n)$。这样保证了当 n 足够大时，$f(n)$ 在一个

常数因子范围内与 $g(n)$ 是相等的，$g(n)$ 是 $f(n)$ 的一个渐进确界。比如 $T(n)=2n+3$，同样也符合 $\Theta(n)$。此时我们称 $f(n)$ 和 $g(n)$ 是同阶的函数。

相应的，还有 Ω 记号，Ω 记号表示函数 $f(n)$ 构成的集合，存在 n_0,c_1,c_2，当 $n\geqslant n_0$ 时，$0\leqslant cg(n)\leqslant f(n)$。换句话说，$g(n)$ 是 $f(n)$ 的低阶函数，或者说 $g(n)$ 是 $f(n)$ 的下界。

如果希望大小关系不包含等于，则还有 ω 和 o 两种记号。其中 $\omega(g(n))$ 表示存在 n_0,c_1,c_2，当 $n\geqslant n_0$ 时，$0\leqslant cg(n)<f(n)$；而 $o(g(n))$ 表示存在 n_0,c_1,c_2，当 $n\geqslant n_0$ 时，$0\leqslant f(n)<cg(n)$。它们与大 O 记号和 Ω 记号类似，只是在大小关系上不包含等于。

小可：嗯，听到这里，我理解了如何进行算法的分析和几种记号表示的含义了。

Mr. 王：另外，很多时候，算法的运行时间并不是稳定的，在算法分析的过程中，我们还要考虑算法运行的最好情况、最坏情况和平均情况。很多算法的最好情况非常好，但平均情况不够理想；而有的算法运行时间的最坏情况复杂度非常高，但平均情况却不错。

这里我们举个例子来说吧。对于一个存放了 100 个整数元素的数组，而且这些整数是无序的，我们要从中找到一个数字 50。这就存在最好情况和最坏情况。

小可：我明白了，如果使用逐个访问数组中每个元素的算法，最好情况是一比较发现第一个元素也就是 $A[0]$ 恰好是 50，那就不用再往后比较了，只进行一次比较就找到了 50；最坏情况就是逐个比较下去，发现最后一个元素是 50，或者最后一个元素也不是 50，则说明数组中不存在 50 这个元素。这时候要对每个元素都进行一次比较，这里有 100 个元素，就进行了 100 次比较。

Mr. 王：那算法的最好和最坏情况复杂度如何呢？

小可：如果有 n 个元素，在最好情况下，可以以常数时间找到我们所要找的元素，也就是 $O(1)$；在最坏情况下，我们要和最后一个元素进行比较才能得出结论，就是要进行和数据规模 n 相关的次数比较，也就是 $O(n)$。

Mr. 王：嗯，对于很多算法来说，用最好和最坏情况都不能够最佳地代表一个算法的复杂度。就拿这个例子来说，用最好情况来代表算法的复杂度显然是不恰当的，因为我们要找的元素往往不能总是第一个元素，如果这些元素是随机分布的，只有 $1/n$ 的概率让其出现在第一个位置上。同理，我们要找的元素恰好出现在最后一个位置上的概率也是 $1/n$，所以说它的运行时间就是访问 n 个元素的运行时间显然也不够公平。因此，很有必要给出一种复杂度，叫作平均复杂度，顾名思义，平均复杂度就是算法运行的平均情况的时间复杂度，既不是最好的，也不是最坏的，而是所有情况的平均值，或者说是在所有情况下复杂度的数学期望，很多时候，平均复杂度能最好地概括一个算法的运行情况。那么，从数组中逐个搜索一个元素的算法的平均情况如何呢？

小可：如果元素是随机分布的，元素出现在数组中每一个位置上的概率就是均等的，所以期望的运行时间应该是访问 $n/2$ 个元素的时间，也就是 $O(n/2)$。不过，这个值好像也是 $O(n)$ 啊。

Mr. 王：嗯，对于这个算法来说，平均情况和最坏情况的复杂度是同数量级的；但对于一些算法来说，最坏情况的复杂度却要比平均情况高一个数量级，用最坏情况去衡量它的复杂度就会将其评价为不够快的算法，这也不够公平。所以对于很多算法来说，我们要考虑它的最好、最坏和平均情况，以便更好地估计一个算法运行的真正时间。

2.3 基础数据结构——线性表

Mr. 王：为了以后的知识描述方便，这里简单介绍一下数据结构的概念。数据结构是一个广泛存在于计算机科学中的概念。曾经有一位计算机界的大师说："数据结构 + 算法 = 程序"。随着计算机科学的发展，虽然现在这个理论被认为不够全面，但也足以说明数据结构的重要性。

小可：这么说，数据结构拥有和算法同样重要的地位了！那么数据结构究竟是什么呢？

Mr. 王：在客观世界中，信息是多种多样的，有数字、颜色、图形、文本、声音等。但是这些信息"本身"并不能直接存储在计算机中，而是要以数据的形式存储在计算机中。比如要描述一个颜色，就可以用显示器输出的红色、绿色、蓝色的值（RGB 值）来表示；描述一张图片，就是用多个这样的 RGB 值或者标识颜色的数据标签来记录的。不论何种类型的信息都要以数据的形式在计算机中进行表示。

不过，这些数据不能杂乱无章地存放在计算机中；否则，不仅损失了信息之间应有的逻辑关系，而且查找起来效率也比较低，所以我们就要试图去发现这些数据之间的一些相互关系，并且利用这些关系将数据组织成一种逻辑结构。这种结构就是数据结构，它研究的就是如何依照这些数据之间的逻辑关系将数据表示在计算机中。对于一组数据，我们不仅要关注它们的逻辑结构，也就是数据之间在逻辑上的相互关系，在实际使用时，还要关注它们的存储结构，也就是说，这些数据在内存（磁盘）中是如何存储的。这都是数据结构研究的范畴。

小可：既然有人说，程序等于数据结构加算法，那么数据结构和算法之间又有怎么样的关系呢？

Mr. 王：这是个很好的问题，计算机中的算法都要对数据进行处理，而既然要处理数据，就要涉及数据如何存储在计算机中，如何能够高效地访问和处理数据，这就需要用到数据结构，所以说算法离不开数据结构。而当我们需要使用、访问、添加、删除、修改存储在数据结构之中的数据时，也要依照数据结构的特点进行操作，而增加、删除、修改、查找这些操作本身就是一种算法，所以说数据结构也离不开算法。

小可：原来是这样，数据结构和算法之间的联系还真是紧密啊。

Mr. 王：下面我们就来介绍一下最基础的数据结构——线性表。

小可：线性表是不是就是这些数据一个挨着一个地线性存储的结构呢？

Mr. 王：是的，线性表是由相同类型的数据按照一定的顺序排成的序列。这是一种非常常见的基础数据结构，使用非常广泛。

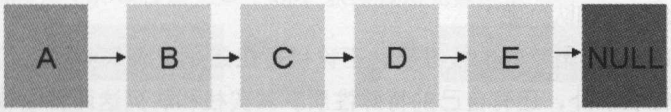
小可：具体的线性表都有哪些呢？

Mr. 王：常见的线性表有链表、数组线性表、栈和队列。

小可：数组我知道！ C 语言和 Java 中就有，将同一类型的数据存储在连续的内存空间里，通过首地址和数组下标可以访问到数组中的每一个元素。嗯，数组果然是符合线性表的定义的。

Mr. 王：是的，数组就是一种典型的线性表。我们学过了算法分析，可以考虑一下，数组只要知道首地址和下标就可以找到一个元素，这意味着能够以 O(1) 的复杂度访问其中的每一个元素，从这个角度来看，数组也是具有其优势的。但是增加和删除其中的元素却比较麻烦，因为如果要在数组中间增加一个元素，需要将后面的元素全都向后移动一个位置，平均情况需要 O(n) 次移动数据，是非常耗时的。前面我们也讨论过，如果数组内容是无序存储的，查找其中某个特定的值就会比较困难，要逐个地去访问比较，需要 O(n) 的时间复杂度。从空间角度来讲，如果我们确定知道元素的数量，那么用数组会比较节省空间；但是对于那些不知道有多少个元素、需要频繁添加和删除的元素集合来说，就需要在定义数组时让它稍大一些，这就造成了空间浪费。

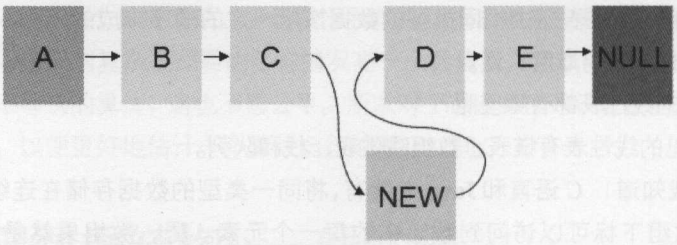
另外一种比较常见的线性结构是链表，它的构建是将元素一个一个地链接起来。访问链表时有一个首地址，我们可以通过这个首地址找到链表的第一个元素，这和数组是一样的；但和数组不一样的地方是，它的元素可能不连续地存储在空间中，而是每个元素的后面都带着一个地址，这个地址指向下一个元素的位置。我们想要逐个访问元素时，就要先找到第一个元素，通过第一个元素后面的地址找到第二个元素，然后再通过第二个元素后面的地址找到第三个元素，依此类推。



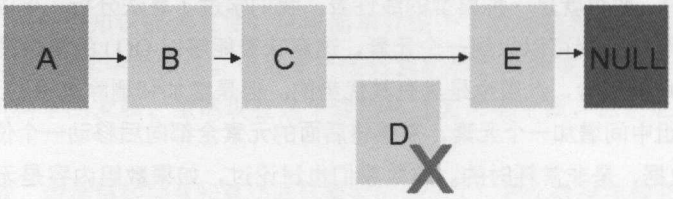
链表的例子

小可：那访问一个元素的时间复杂度就不是 O(1) 了，从平均情况来讲，访问一个元素的时间复杂度就是 O(n)。

Mr. 王：很好，但是链表也并不是没有优点。链表的优点是，在一个特定的位置添加元素，并不需要移动任何元素的位置，只需要将其前面元素后面的地址指向新元素，再将新元素后面的地址指向前面元素原来的后面地址就可以了。删除也是一样，只需要将待删除元素前面元素的指针指向删除后面的元素，然后释放掉被删除元素的内存即可。



链表的插入操作



链表的删除操作

小可：这个时间复杂度是 O(1)。

Mr. 王：这说明链表适合用于那些需要频繁进行添加、删除的元素集合。另外，从空间角度来讲，链表的长度相对比较灵活，它不像数组，即使数组中某一个位置是空的，没有元素，也要占用预先申请的内存空间；而链表每有一个元素，就占用一个元素和它后继节点地址的存储空间，但相对于元素数量变化幅度比较大的元素集合来说，链表的存储更加高效。

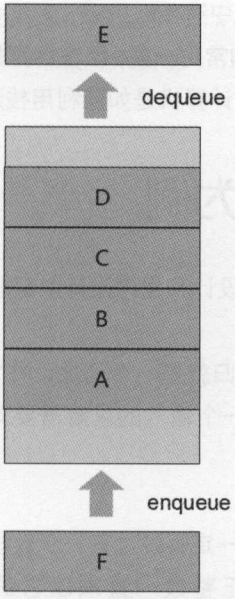
小可：嗯，果然是互补的两种结构，要根据实际情况选择适合的结构来存储。

Mr. 王：这里还有两种非常重要的线性结构要介绍，这两种结构非常简单，但却是很多算法的基础和组成部分，它们就是栈和队列。

小可：那什么是栈和队列呢?

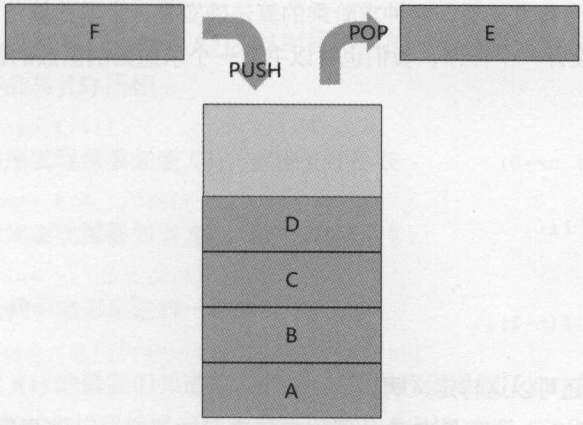
Mr. 王：栈和队列都是特殊的线性表，不论用数组还是链表来实现它们都是可以的。但是它们除了是线性表以外，还有自己的特殊性质。其实栈和队列这两种结构在生活中都可以见到。

先说队列吧。平时我们在各种公共场所排队时，如果将每一个人都看作一个元素的话，那么组成的就是一个队列。队列的特点就是，每一个新加入的元素，都要放在整个队列的最后面；而当一个元素要离开队列时，总是从队列的最前端离开。这就是说，先进入队列的元素先离开队列，后进入队列的元素后离开队列，总结起来就是 FIFO 策略（先进先出策略）。队列的常见操作有入队列（enqueue，让一个元素进入队列）、取队首（front，获得队列最前端元素的值）和出队列（dequeue，让队首的元素离开队列）等，这些算法都非常简单。



小可：嗯，队列还是很直观的。那什么是栈呢？

Mr. 王：栈的特点和队列正好相反。用生活中的例子来说，就像是一个书箱子，箱子只有一个朝上的开口，我们可以把书从上面放进箱子，但是想要取书时，也只能从箱子的顶端取走，不能直接从箱子的中间和底部取书。所以，先进入箱子的书就会最后离开箱子。栈只有一个访问端，元素的出入都只能通过这一个端，这样我们也可以分析出，先进入栈的元素会被放在栈的最底端，也就是说，它会最后离开栈。总结起来就是 LIFO 策略（后进先出策略）。对栈的常见操作有入栈（push，将一个新元素放在栈顶的位置）、出栈（pop，将栈顶的元素弹出栈，露出下一个元素作为栈顶）和取栈顶元素（top，获得栈顶元素的值）。



栈的示意图

小可：嗯，我懂了。可是栈在计算机中有什么用呢？

Mr. 王：看起来队列在生活中似乎更加常见一些，但在计算机中，栈的应用却是非常广泛的，我们可以通过一个非常经典的例子来介绍计算机是如何利用栈这个结构的。

2.4 递归——以阶乘为例

Mr. 王：我们介绍一个在计算机算法设计和程序设计中都非常常见的概念——递归。

小可：什么是递归呢？

Mr. 王：从程序设计的角度来说，递归就是一个函数，在它的定义中调用了它本身。从算法的角度来说，递归就是一个算法对于一个输入的求解需要对这个算法在更小输入上求解的情况。

小可：这个说法听起来有点复杂啊。

Mr. 王：我们举个例子来说明吧。你一定听说过有一个数学概念叫作阶乘。

小可：我知道，阶乘就是把一个正整数一直乘以它的值减 1，直到乘数为 1，比如 5!=5×4×3×2×1。推广到 n 的情况就是 $n!=n\times(n-1)\times(n-2)\times\cdots\times3\times2\times1$（特殊的，0!=1）。

Mr. 王：在计算机中求解一个数的阶乘，就可以利用递归。因为阶乘具有一个很有意思的特征，就是：$n!=n\times(n-1)!$。假如我们把阶乘定义为 $f(n)$ 的话（也就是 $f(n)=n!$），就有 $f(n)=n\times f(n-1)$。

小可：哦，从阶乘的定义来看，就是我们想知道 $f(n)$，就要知道 $f(n-1)$ 是多少，推广下去，想知道 $f(n-1)$ 就要知道 $f(n-2)$，一直到 $f(1)$。

Mr. 王：从递归的定义来看，求阶乘这个算法是不是正好符合求对于一个输入 n 的解，需要求取这个算法在一个更小的输入 $n-1$ 上的解，而对于 $n-1$ 的解需要知道去求取 $n-2$ 的解。

小可：嗯，从这个角度来看，这种求阶乘的算法确实是一个递归算法。

Mr. 王：如果要设计一个程序，我们也可以书写一个求递归的函数的伪代码：

```
int f(int n)
{
    if (n==1 || n==0)
    {
        return 1;
    }
    else
        return f(n-1);
}
```

小可：原来函数还可以这样定义啊。

Mr. 王：是的，C/C++ 语言是非常典型的支持递归的语言。一些早期的语言不支持递归，

不过现在很多程序设计语言都支持递归算法的设计。虽然所有的递归算法都可以设计成非递归的版本，比如阶乘，我们可以用一个循环来实现：

```
int f(int n)
{
    if (n==1 || n==0)
    {
        return 1;
    }
    else
    {
        for (int i=1;i <= n;i++)
        {
            int ans=1;
            ans *=i;
        }
        return ans;
    }
}
```

但是递归往往可以更加直观地表达算法的思路，这是非常有利于算法实现和程序设计的。不过有一点需要注意，设计不好的递归算法是非常容易出现无限循环的，在设计递归算法时，一定要设计递归的终点。比如在阶乘中，我们必须指定递归最终会达到的结果 $f(1)=1$；否则程序就会一直执行下去，直到内存溢出。

小可：嗯，我懂什么是递归了，但是这和栈有什么关系呢？在递归算法中也没有发现栈的存在啊？

Mr. 王：递归算法和栈的联系非常紧密，虽然在递归程序中我们并没有直接定义出一个栈，但程序运行的内部却会帮我们生成一个栈，这对于递归算法的运行是必要的。现在我们就以阶乘为例来剖析递归算法是如何运行的。

比如我们要求 5 的阶乘，也就是 $f(5)$。这时程序内的一个栈空间会开始工作，这个空间叫作函数调用栈。程序会将 $f(5)$ 压栈：

```
Call stack：[top= f(5)]
```

求解 $f(5)$ 时，程序发现需要知道 $f(4)$，就把 $f(4)$ 压栈：

```
Call stack：[top= f(4)][f(5)]
```

求解 $f(4)$ 时，程序发现需要知道 $f(3)$，就把 $f(3)$ 压栈：

```
Call stack：[top= f(3)][f(4)][f(5)]
```

依此类推，最后栈中会形成这样一种情况：

```
Call stack：[top= f(1)][f(2)][f(3)][f(4)][f(5)]
```

此时，程序发现 $f(1)$ 的值我们知道了，$f(1)=1$。所以我们得到了 $f(1)$ 的解，$f(1)$ 这个函数返回 1，已经解决的问题或者说已经返回的函数就会弹出调用栈：

```
Call stack：[top=f(2)][f(3)][f(4)][f(5)]
```

然后，程序发现 $f(1)$ 我们知道了，$f(2)$ 也就知道了，$f(2)=2\times f(1)$。$f(2)$ 返回了值 2，$f(2)$ 得到解决之后再将 $f(2)$ 移出栈：

```
Call stack：[top=f(3)][f(4)][f(5)]
```

依此类推，程序发现 $f(2)$ 我们知道了，$f(3)$ 也就知道了，$f(3)=3\times f(2)$。$f(3)$ 返回了值 6，相应地，$f(3)$ 得到解决之后再将 $f(3)$ 移出栈：

```
Call stack：[top=f(4)][f(5)]
```

不断地执行下去，就能够得出 $f(5)$ 的值为 120，此时栈空，程序结束。

不难看出，在运行递归程序时，栈一直在工作。因为我们调用函数的嵌套关系恰好满足先到的问题后得到结果、先调用的函数最后返回这样的关系，所以语言的设计者们就利用这一点，用栈结构来表示函数的调用关系。

小可：原来是这样，虽然看不见，但栈一直存在于我们设计的递归和函数调用程序之中。

Mr. 王：是的，栈这种看似简单的数据结构，其实应用是非常广泛的。

这里再谈谈以递归实现算法的缺点。递归程序虽然能够非常有效地表达程序的思路，使得程序的书写变得非常简洁，易于理解，但它的运行速度和执行同样工作的非递归版本相比往往是比较慢的，如果对程序的执行效率有要求，则可以将递归版本重写为非递归的。另外，递归程序在实际的执行过程中执行了多少层递归是不容易预测的。我们知道，前面提到的调用栈也是在计算机的内存空间中，如果递归的层次非常深，就会导致调用栈占用的内存空间被占满，无法继续下一层递归的运行，这就是很多人说的栈溢出或者说“爆栈”，栈溢出会导致程序运行崩溃，所以递归也并不是十全十美的。还是一定要对程序的运行环境进行评估，选择设计递归或者非递归版本的程序。

第3章　何谓大数据算法

Mr. 王：下面我们就来谈谈大数据算法与一般算法的区别和联系。

小可：好。

Mr. 王：前面我们讲了如何评价一个算法，在相对比较小的数据规模下，我们往往可以接受多项式时间算法。但是当数据量很大时，很多小数据量上我们能够在可以接受的时间内解决问题的方法，也都变得不再可以接受。虽然有些算法是多项式算法，但是它的高阶项指数却是非常大的，导致当数据规模大起来时，它的增长速度会变得非常快。对于较大的数据量，资源约束和时间约束都变得相对很苛刻，我们要对可以接受的时间界限进行重新思考。

小可：那在大数据上比较好的算法是什么样的呢？

Mr. 王：**大数据算法**是在给定的资源约束下，以大数据为输入，在给定的时间约束内可以生成满足给定约束结果的算法。

对于大数据而言，访问全部数据是很费时的，所以大数据算法有时需要采取读取部分数据的办法，也就是设计时间亚线性算法。而且数据往往在内存中也存不下，数据要存储在磁盘上，所以要考虑设计外存算法；或者是采取读取部分数据的办法，设计空间亚线性算法。

小可：亚线性算法有一种抽样的感觉，不访问全部数据，而是尝试选择部分数据来代表全部数据。

Mr. 王：没错。如果单台计算机不能保存所有的数据，或者一台计算机的计算资源不足以在给定的时间内解决问题，则还要引入多台计算机进行并行处理，让它们的 CPU、内存、磁盘都参与到问题的解决之中，这时候还要设计并行算法。甚至当计算机的能力不足以对某些问题进行处理，或者对某些问题的处理不够好，而这些问题的某一部分恰好是人类非常擅长做的工作时，还可以引入人工参与到问题的解决之中。

小可：引入人工来帮忙，这倒真是神奇啊！我还以为计算机算法都是一定要计算机来执行的呢，大数据算法还可以不是完全由计算机执行的啊？

Mr. 王：嗯，不仅如此，大数据算法还可以不是内存算法，很多时候需要磁盘参与到海量数据的存储之中；可以不是精确算法，很多时候得出精确解的代价过大，大数据算法就以得出一个足够让我们满意的近似解来谋求更高的计算效率；可以不是串行算法，在很多常见的大数据问题求解中，引入多台计算机参与到其中，发挥它们的各种计算资源的作用以提升问题的解决速度；甚至可以不是仅仅由计算机来执行的算法，在某些特定的情况下，有很多问题由人工来解决会比由机器来解决更容易、更准确或者更高效一些。从这些方面来看，大数据算法的设计和分析与传统的经典算法有着很大的区别。

另外，在大数据算法中常用的算法设计技术有：精确算法设计方法、并行算法、近似算法、随机算法、在线算法 / 数据流算法、外存算法、面向新型体系结构的算法、现代优化算法等。在大数据算法设计之后，与经典算法一样，依然要对算法进行分析。但在分析大数据算法时，我们需要研究的也不仅仅限于时间、空间复杂度的分析和优化。对于一些不能得出精确解的算法，还要对结果质量进行分析，看看在我们可以接受的时间范围内得出的近似结论是不是足够达到要求；对于要借助磁盘存储的算法，还要考虑磁盘的 IO 复杂性。有时候大数据算法会运行在比如无线传感器节点这样的对电池电力有较强限制的终端上，我们还要分析算法运行消耗的能量是不是很大，这时还要进行能量复杂度分析；如果我们使用的是一个分布式系统，整个系统架构在网络上，要依靠各个节点的频繁通信来实现，那么还要考虑系统的通信复杂度。这些特点也使得大数据算法的分析变得相对复杂一些。

这些概念现在听起来也是一头雾水吧？

小可：嗯，的确。

Mr. 王：在以后的课程中，我会把大数据算法所涉及的内容讲解给你。以后我会给你讲讲大数据算法中的亚线性算法、外存算法、并行算法、众包算法，这些都是大数据算法中的核心算法。时间不早了，我们先下课吧。

小可：那太好了。那就明天再见了，老师。

第 2 篇　理论篇

第4章 窥一斑而见全豹——亚线性算法

4.1 亚线性算法的定义

Mr. 王：从今天开始，我们正式讲解大数据算法的内容。首先谈谈关于亚线性算法的问题。

小可：我记得前面提到过亚线性算法，就是复杂度低于输入规模的算法。

Mr. 王：我们给出一个严格的定义，还是设输入规模为 n，那么亚线性算法就是指时间、空间、通讯、能量等复杂度为 $O(n)$ 的算法。

小可若有所思，说：如果输入规模为 n，而算法的复杂度还要低于 n，这是不是说我们不能保存所有的数据，或者不能访问所有的数据呢？

Mr. 王：是的。只有这样才能实现亚线性的要求。

小可：可是，如果访问不到所有的数据，对于很多问题我们是得不到正确答案的啊。比如有一组规模比较大的数据，我们要求它们的中位数，如果不访问所有的数据，得到的结果就有可能是错误的啊。

Mr. 王：对于这样的答案我们不能简单地称之为正确解和错误解，而是称之为**精确解**和**近似解**，在大数据算法中解决问题的重要思路就是**近似**。由于在规定的时间内和计算条件下，我们得到精确解的时间太久，所以采用近似的方法来得到一个“差不多”的答案。这样的答案的误差在我们可以容忍的范围内，能够满足应用的需求就可以了。在小的数据集合上，我们时常需要的是精确解，而当数据量很大时，得到精确解的开销也会变得大到不能接受。因此，我们

求近似解以换取更高的效率。近似是亚线性算法的核心思想。

小可：原来是这样。

Mr. 王：亚线性算法也可以分为空间亚线性算法、时间亚线性计算算法和时间亚线性判定算法。下面我就这几类问题分别举个典型的例子来看看亚线性算法是如何解决问题的。

4.2　空间亚线性算法

4.2.1　水库抽样

Mr. 王：首先我们看一个经典问题：水库抽样。这是一个典型的空间亚线性算法。考虑这样一个问题：很多时候我们要在大宗数据中进行一个均匀的抽样，这在实际的生活中是非常常见的，但是在实际情况下，有时候我们要处理的数据量太大，以至于只能让这些数据从我们的面前“流过”一次。这就好比数据从一个生产线或者流水线上源源不断地到来，当来到计算系统中时，我们可以对其进行处理，但是对其进行过处理之后，或者由于能存储这些数据的能力有限，我们不得不将其从内存中清除出去，这样就再也不能访问它了。

水库抽样问题的要求是，每一刻所取到的样本，就是前面已经“流过”的全部数据的均匀抽样。你来根据我们前面提过的分析问题的方法，说一说问题定义。

小可想了想，说：

输入：一组数据，其大小未知。

输出：这组数据的 k 个均匀抽样。

要求：

a．仅扫描数据一次；

b．空间复杂性为 $O(k)$；

c．扫描到前 n（$n>k$）个数据时，保存当前已扫描数据的 k 个均匀抽样。

Mr. 王：说得不错，对于这个问题，我们给出下面的算法：

（1）申请一个长度为 k 的数组 A 保存抽样（此处的数组下标以 $[1,k]$ 表示，对于 $[0,k-1]$ 的实际操作情况，可以非常容易地进行替换）。

（2）保存首先接收到的 k 个元素。

（3）当接收到第 i 个新元素 t 时，生成 $[1,i]$ 间随机数 j，若 $j \leqslant k$，则以 t 替换 $A[j]$。

算法的关键在于第 3 步：每当新到来一个元素 t 时，都生成一个随机数，一旦随机数落在了数组范围之内，就用新元素把它替换掉。

小可：这个算法看起来倒是挺简单的，用一个随机数来决定是不是进行抽样替换，可是它

真的能够满足均匀抽样的需求吗?

Mr. 王：嗯，接下来我们就来证明这个算法的正确性。你先说一说，均匀采样应该满足什么条件?

小可：在本题目的条件中，对于任意一个元素 i，它被选入样本的概率均为 k/n。

Mr. 王：好，那么我们只需要证明该算法满足这个要求就可以了。首先，对于每一个新到来的元素 i，它是以 k/i 的概率被收入抽样集合的，这是因为生成的随机数范围是 $[1,i]$，而当数字小于等于 k 时，它会被替换进数组 A。

当第 i+1 个元素到来时，i+1 被替换进来的概率就是 k/i+1，而此时，前一个元素 i 被从中替换出来的概率是 $1/k$。这两个值的乘积就是当第 i+1 个元素到来时前面的 i 被替换出来的概率，其值为 $1/(i+1)$，那么 $1-1/(i+1)$ 就是 i 没有在 i+1 到来时被替换出去的概率。当 i+2、i+3 等这些元素到来时，其计算过程和 i+1 是同理的。

如果元素 i 被选入集合中，并且在后面所有的替换过程中，每一次替换都没有被替换出去时，它就是我们选出来的样本，那么元素 i 在样本中的概率应该是多少呢?

小可：

$$\frac{k}{i}\times\left(1-\frac{1}{i+1}\right)\times\left(1-\frac{1}{i+2}\right)\times\cdots\times\left(1-\frac{1}{n}\right)=\frac{k}{n}$$

Mr. 王：这就是说，对于任意元素 i，其被选入样本的概率均为 k/n。也就是说，它符合随机抽样。

小可：原来随机决定了替换的结果，还真的能保证抽样的均匀性。

Mr. 王：讨论过算法的正确性，可以确定这个算法的执行结果是正确的，但是我们希望它是一个空间亚线性算法，所以还要分析它的空间复杂度是不是满足亚线性这一要求。你来分析一下，这个算法的空间复杂度如何?

小可：不论“流动”来了多少个数据，我们只需要保存 k 个数据作为样本就可以了，其余的计算空间都是常数开销，那就应该是 $O(k)$。

Mr. 王：显然，k 是小于 n 的。也就是说，我们的这个算法在正确的前提下，对于输入规模 n，做到了 $O(k)$ 的空间复杂度，而 $O(k) \in o(n)$，也就表明，它是一个空间亚线性算法。

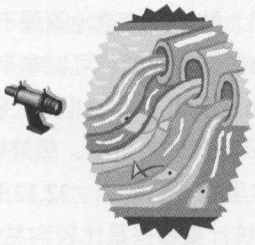
小可：我懂了。

Mr. 王：这里也为我们设计空间亚线性算法提供了一个思路，就是不能去尝试保存全部的数据。我们要进行亚线性的均匀抽样，不能把所有的数据都保存下来，再去做均匀抽样，这样势必会造成 $O(n)$ 的空间复杂度。我们要考虑，能不能在只保存大数据中的一个小集合的情况下，来达到问题的要求。在水库抽样问题中，虽然我们采用的是空间亚线性算法，但是得到的依然是精确解。

4.2.2　数据流中的频繁元素

Mr. 王：我们再来讲一个例子，数据流中的频繁元素。我们先来说说大数据的数据流模型。

小可：数据流，是流动的数据的意思吗？和我们前面说的水库抽样是不是很像？



数据流

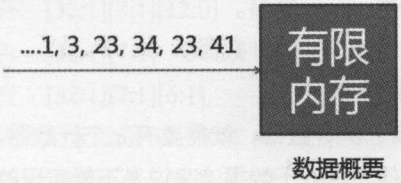
Mr. 王点点头，说：嗯，其实水库抽样也是以数据流的思想来处理的。顾名思义，数据流意味着数据在流动，数据会不断地到达计算系统进行处理，这意味着一个数据只能被扫描一次，一旦处理过或者在内存中被放弃，就不能再访问了。你想想看，这在复杂度上意味着什么？

小可想了想，说：超过 O(n) 的算法肯定是不行的，只能寻找亚线性算法了。

Mr. 王：没错，而且数据流模型的数据是源源不断到来的，比如我们使用一个传感器进行感知，它就会按照一定的采样频率进行数据采集。其实传感器获得的数据就是一个典型的数据流，如果传感器一直在工作，我们就会得到源源不断的数据。但是不管是传感器使用的内存还是其他存储介质都不是无限的，这是符合客观现实的。所以对数据流进行处理，要求我们所使用的内存必须是亚线性的，最好是和数据量无关的。

小可：就像水库抽样一样吧，内存中随时保存着的都是对前面数据流的一个均匀抽样，而且所使用的内存有限，不论来了多少数据，都只保存 k 个，也是与数据量无关的。

Mr. 王：很好，这样用于概括数据的数据结构叫作数据概要，我们用有限的、与数据量无关的内存空间来维护这个数据概要，就是要对相关性质的当前状态做出一个有效估计。



数据流模型

我们说数据流模型是适用于大数据的，因为它仅顺序扫描数据一次，而且它的内存是亚线性的。数据流通常是来自某个域中元素的序列，$<x_1, x_2, x_3, x_4, ...>$。

小可：这个序列里为什么没有末项 x_n？

Mr. 王：因为数据是源源不断的，所以没有末项。好了，我们来总结一下数据流模型的特点：

（1）数据流通常是来自某个域中元素的序列，$<x_1, x_2, x_3, x_4, ...>$。

（2）数据量是远大于内存容量的，这意味着无法将所有的数据都放进内存中。内存的规模一般为 $O(\log^k n)$ 或者 $O(n^\alpha)$，显然 $\alpha<1$。

（3）要快速处理每一个数据，因为数据会快速地源源不断地到来，这也是很必要的一点。

小可：嗯，那我们将大数据视为数据流模型，可以拿来计算什么呢？

Mr. 王：应用就有很多了，有 sum（求和）、max（最大值）、min（最小值）、count（计数）、avg（平均数）这样的基本统计量，还有比如中位数、频繁项、分析、挖掘、预警等。

我们还是举个具体的例子吧，这里有一组数字：32,12,14,32,7,12,32,7,6,12,4。

Mr. 王：你说说看，前面提到的统计量哪些是比较容易计算的？

小可：前面讲解选择排序时，讲过求 min 的方法，我觉得那个方法同样适用于数据流，只要拿出一个变量来保存当前的最小值，发现比它更小的就将其替换掉；max 与之同理。求和 sum 也是很容易的，只要用一个变量来保存当前的加和，每到来一个数据，就把它加进这个变量里。至于 count，初始化为 0，然后每到来一个数据，就把它加 1。如果要求平均数 avg，就同时维护 sum 和 count 作商。

Mr. 王：很好，可以看出，这样的数据概要是单个值，同时它也是一个可合并的值。在这里，“可合并”就是指两部分数据流的数据概要可以直接通过诸如累加和比较这样的操作合并成整个数据流的概要。

小可：嗯，比如我们要求 200 个数据的最大值，前 100 个数据的最大值和后 100 个数据的最大值的较大者，就是 200 个数据的最大值。

Mr. 王：现在我们来处理一个复杂一点的问题——频繁元素。

为了研究方便，我们用 n 来表示不同元素的个数，用 m 来表示元素的总个数。

你先来想一想，如果要求出一个精确解的话，可以用什么方法？

小可：可以用这样的方法：

（1）为每一个单独元素设置一个计数器。

（2）当处理一个元素时，增加相应的计数器。

Mr. 王：这样做有什么问题吗？

小可：如果每个元素都有自己的计数器，就需要有 n 个计数器，这样没有满足 $o(n)$ 的要求，相当于内存需要存储所有的数据，这对于数据流来说是不能实现的。

Mr. 王：嗯，所以我们提出如下的方法：

（1）处理一个新到来的元素 x 时

（2）If 已经为其分配了计数器，增加之

（3）Else If 没有相应的计数器，但计数器个数少于 k，为 x 分配计数器 k，并设为 1

（4）Else 所有的计数器值减 1，删除值为 0 的计数器

这个算法称为 Misra Gries(MG)算法。第一种情况，如果内存中已经有新到来元素的计数器，则只需要将其值加 1 即可；第二种情况，如果还没有为新到来的元素提供计数器，并且内存没有被填满时，则可以为这个元素的计数器开辟新的空间；第三种情况，当新到来的元素没有被分配计数器，同时内存中的计数器个数已经达到了 k 个，也就是分配的内存空间已经被填满时，则将所有的计数器值减 1，删除值为 0 的计数器，此时内存中就重新有位置了，我们再为这个新到达的元素分配一个计数器即可。当然，别忘了要将其置为 1。

我们用前面的这组数字举个例子：32,12,14,32,7,12,32,7,6。

假设内存中有 3 个存放计数的空间。

前 3 个数据进入内存时，都符合情况二，将它们加入内存中。

抵达数据：32　　内存：[32:1]

抵达数据：12　　内存：[32:1][12:1]

抵达数据：14　　内存：[32:1][12:1][14:1]

第 4 个数据 32 到来时，将 32 的计数值加 1。

抵达数据：32　　内存：[32:2][12:1][14:1]

当第 5 个数据 7 抵达时，符合情况三，也就是频繁元素统计的大数据处理的关键，我们将所有的计数器值减 1，并删除那些值为 0 的计数器，然后为新到来的 7 建立计数器，并置为 1。

抵达数据：7　　内存：[32:1][12:0][14:0]

内存：[32:1]

内存：[32:1][7:1]

然后是 12、32 和 7，分别属于情况二和情况一。

抵达数据：12　　内存：[32:1][7:1][12:1]

抵达数据：32　　内存：[32:2][7:1][12:1]

抵达数据：7　　内存：[32:2][7:2][12:1]

当 6 到达时，内存已经被填满，我们执行算法的第三种情况。

抵达数据：6　　内存：[32:1][7:1][12:0]

内存：[32:1][7:1]

内存：[32:1][7:1][6:1]

使用该算法求出的几个频繁元素就是 32、7 和 6。就原数据来说，最频繁的 3 个元素分别是 32、7、12。我们成功地找出了 32 和 7，虽然没有找出最频繁的 3 个元素，但整体来说已经是一个不错的结果了。如果只需要最频繁的元素，那么该算法已经在这组数字中找出了 32 这个最频繁的元素。不过在最后对频繁元素的计数值一般是不准确的，所以还要对它的计数进行分析，估计它所记录的数值误差如何。当我们使用近似手段处理问题时，一定要对近似解进行误差分析，研究所得到的近似解是否在我们可以接受的误差范围之内，误差太大的近似解也同

样达不到解决问题的目的。

你说说看，这个近似算法是高估了频繁元素的数量还是低估了？

小可：内存中的频繁元素的计数器不断地由于内存被占满而被削减，显然是低估了。

Mr. 王：没错，我们不能仅仅知道结果是低估了准确值，最好还要能根据算法分析出究竟低估了多少。想要知道低估了多少，我们首先要考虑的就是一个计数器被减小了几次。这就需要我们考虑到在整个算法的执行过程中，执行过多少个减少计数器的步骤。假如把整个结构的权重（也就是计数器的和）记作 m'，整个数据流的权重（全部元素的数量）记作 m。每当计数器需要降低时，由于内存中有 k 个计数器，我们也就减少了 k 个计数，但是这时新到来的元素 x 并未计入内存中的计数器，它的到来只是标志着该削减计数器了，所以我们少加了 k+1 个计数器。因此，最多有$\frac{m-m'}{k+1}$个减少步骤。也就是说，估计和真实值最多相差$\frac{m-m'}{k+1}$。如果数据流的元素总量远大于$\frac{m-m'}{k+1}$值，我们可以得到一个好的频繁元素的估计。

可以看出，错误的界限是与 k 成反比的。你说说看，这说明什么？

小可：k 是内存中计数器的个数，也就是程序使用内存的大小，这说明内存越大，结果就越准确。

Mr. 王：嗯，这也是符合客观规律的。而且我们可以利用数据概要来计算错误的界限，只需要记录 m、m' 和 k 就可以了。

不过不难看出，如果数据集合中每个元素的数量都相差不多的话，这个算法求出的结果会具有很大的随机性，好在我们一般需要处理的数据都满足 Zipf 法则。

Zipf 法则：典型的频率分布是高度偏斜的，只有少数频繁元素，最多 10% 的元素占元素总个数的 90%。这个定律说明，只有少数的元素是大量重复出现的，而绝大多数元素的出现是不频繁的。

根据 Zipf 法则我们知道，频繁元素的种类只有少数，而其数量往往是非常大的，在算法执行的过程中，不断地削减内存中的计数器对于频繁元素最终被保留在内存里不会有太大程度的影响。

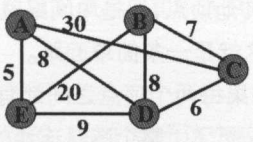
4.3 时间亚线性计算算法

4.3.1 图论基础回顾

Mr. 王：我们再讲一个时间亚线性算法——平面图直径的求解。平面图是图论中的一个概念，在大数据算法的很多地方都会涉及图的相关内容，所以这里我们还是要回顾一下图论的知识。

首先来看看什么是图。其实很多文献对图的定义都是不尽相同的，但是整体的描述差异不大。

我们一般用 $G(V,E)$ 来表示一个图，其中 G 表示这个图，V 是这个图的顶点集合，E 是这个图的边集合。



图的例子

比如在上面的图 $G(V,E)$ 中：

V={A,B,C,D,E}

E={(A,E),(A,D),(A,C),(B,E),(B,D),(B,C),(D,C),(D,E)}

整体上图可以分为两种：有向图和无向图。这里的有向和无向是相对边来说的。在无向图中，边是没有方向的，连接顶点 u 和 v 的边可以记为 (u,v)，当然也可以记为 (v,u)。由于边是没有方向的，所以这两种表示法表示的是同一条边。在无向图中，每一条边都是双向可达的，如果有边 (u,v) 存在的话，那么 u 是 v 的邻居，v 也是 u 的邻居。与 u 直接相连的边的数量，叫作 u 的度数。

而在有向图中则不然，每一条边都是有方向的，也就是说，(u,v) 这条边表示的是从 u 指向 v 的一条边；而 (v,u) 这条边表示的是从 v 指向 u 的一条边。它们都是单向可达的。如果仅有 (u,v) 这条边存在，u 可以通过 (u,v) 到达 v，但 v 却不能通过这条边到达 u，即 (u,v) 和 (v,u) 是两条不同的边。以 u 为起点的边，叫作 u 的出度；以 u 为终点的边，叫作 u 的入度。在图形表示中，我们使用带有箭头的线来表示有向边。

我们使用的图多数都是加权图。在加权图中，有的是边加权，也就是说，边不仅仅是一条边，在边的上面有一个权重，这个权重也可以叫作边的长度，在边不加权的图中，我们一般认为边的长度为 1。还有的是图的顶点具有一个权值。当然，也有顶点和边均具有权值的加权图。

小可：我想一些城市的互联关系，或者说地图就可以抽象成一个加权图吧，图的边权用来表示两个城市之间的距离。

Mr. 王：没错。就用交通地图来举个例子，我们想从一个城市到达另一个城市，而这两个城市之间并没有一条直接相连的公路。我们会选择途经另一个城市，所以就有了路径的产生。如果从一个顶点 u 到另一个顶点 v 中间途经数个顶点 $w_1,w_2,w_3,\cdots$，并且这些顶点之间的边都存在的话，我们称 $<u,w_1,w_2,w_3,\cdots,v>$ 是一条路径。

小可：嗯，这在现实中也是非常普遍存在的。

Mr. 王：我们定义路径的长度，为途经的边的个数。如果中间的那些顶点 $w_1,w_2,w_3,\cdots$ 没有重复的，我们称之为简单路径。如果 u 和 v 是同一个顶点，并且至少经过一条边的话，我们称这条路径是一个回路。

小可若有所思，说：如果 u 本身有一条边指向自己，就是有一个圈，这样也是回路吗？

Mr. 王：虽然没有经过任何一个其他顶点，但是中间经过了一条边，它也是一条回路。相应的，如果回路中没有出现重复的顶点，这就是一条简单回路。

Mr. 王：另外，在无向图中，如果每两个顶点之间都有一条路径，我们称它是连通图。

小可：这样每个顶点就都连在一起了，整个图是连通的。

Mr. 王：几个可达的顶点之间构成的最大的集合，称作连通分量。这个最大的集合是指，如果几个点之间是连通的，只要再添加图中的任何一个顶点就都不再连通。连通分量是一个图的子图。还有一种判定连通图的方法，就是如果一个无向图只有一个连通分量的话，那么它就是连通的。

小可：嗯，在无向图中是这样的，那么在有向图中又如何呢？

Mr. 王：由于有向图的边是有方向的，所以存在这样一种情况，就是虽然两个顶点是有一条边“连着”的，但是却是单向可达的。在这种情况下，我们不能说它是连通的。在有向图中，如果图中每对顶点都互相可达，我们才能认为它是“连通”的，称作强连通图。

小可：的确，相互可达才能达到我们判定它连通这个目的。

Mr. 王：相应的，几个可达的顶点之间构成的最大的集合，称作强连通分量。这与无向图类似，只是必须要注意，对于有向图的连通，我们必须要考虑相互连通这个问题。

Mr. 王：还有一个很重要的问题，就是图在计算机中的表示。虽然我们看到的图边和点等都是非常直观的，可以画成一个圆圈里带一个数字表示顶点，用一条带有数字的线段或者箭头来表示边，但是在计算机中，显然不能用这种方式来存储它。

小可开玩笑地说：要是把图存成图片，那可太占空间了，而且还不容易读取上面的数字。

Mr. 王：是啊，图已经是对现实世界的一个抽象了，在计算机中我们要对其进行进一步的抽象。你想一想，图由哪两部分组成？

小可：边的集合和顶点的集合。

Mr. 王：在手绘的图中，对于顶点的位置和表示边的线段的长度都是没有任何影响的。对于顶点，我们只关注它的编号（ID）和权值；对于边，我们只关注它连接的两个顶点和权值。当然，对于有向图来说，还有方向。

小可：这是不是意味着，我们只需要存储这些信息就可以了？

Mr. 王：是的。不仅如此，我们还希望这些边和点的集合可以被更高效地发现，比如举出一个顶点，就可以很快地找到它的邻居们。所以直接存储所有的边和顶点查询效率不够高，因此计算机工作者们选取了邻接矩阵和邻接表。

小可：那什么是邻接矩阵呢？

Mr. 王：邻接矩阵是这样的，它是一个方阵，行和列这两组表头分别是所有顶点的 ID。比如一个图有 A,B,C,D,E 这些节点，我们就在行表头记 ABCDE，相应的，也在列表头记 ABCDE，这样就有了所有的节点。如果这些节点还有权值，那么就记在另一张表中。实际存储在计算机中时，我们会用一个二维数组来表示，其中 A,B,C,D,E 这些字母用数组下标 0,1,2,3,4 来表示。

小可：那么如何来表示一条边呢？

Mr. 王：数组内存储的数据还是空的，我们就用这个数据域来表示边。假如有一条有向边 AB，它的权值为 5，我们就将数组 $G[0][1]$ 这个位置填充数据 5 即可，对于权值为 6 的边 BC，$G[1][2]=6$。相应的，如果有一条有向边 BA，它的权值为 4，我们就将 $G[1][0]$ 填充为 4。

	0	1	2	3	4
0		5			3
1	4		6	4	
2					
3		3			5
4			1		

邻接矩阵的例子

小可：那么如何表示无向边呢？

Mr. 王：在邻接矩阵的表示中，一般不去区分有向图和无向图。无向图的表示方法和有向图是一致的，只不过在无向图中，对于长度为 3 的无向边 AB，我们将 $G[1][0]$ 和 $G[0][1]$ 的值都改为 3 即可。在这里，其实一条无向边可以看作，两条方向相反、权值相等、连接相同两个顶点的有向边。

	0	1	2	3	4
0		3		4	
1	3		6		
2		6		2	
3	4		2		
4					

无向图的邻接矩阵

小可：那些没有边的数据域呢？

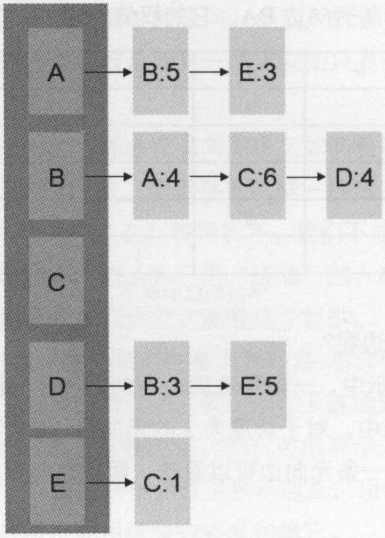
Mr. 王：一般来说，我们会用权值来表示两个顶点的距离。如果没有边，那么这两个点之间的距离可以看作是无穷大，在实际应用中，我们会用一个很大的数来表示它。对于每个顶点到自己的距离，一般记作 0，比如 $G[0][0]=0$，这样可以方便很多算法的处理。另外，对于无权的图，我们将边的权值视作 1，这样方便计算无权图中路径的长度，也就是经过边的数量。

小可：可是邻接矩阵占用空间很大啊，不论两个顶点之间是不是真的有一条边，我们都要用一个数来存储。这岂不是很浪费空间吗？

Mr. 王：所以邻接矩阵更加适合用来存储稠密图，图中的边越多，浪费的空间就越少。

小可：对于那些比较稀疏的图，怎么办呢？

Mr. 王：这就要使用另一种存储结构——邻接表。邻接表比较适合用于存储稀疏的图。邻接表是一个链表的集合，链表所有的表头代表一个节点。比如前面的例子有 A,B,C,D,E 这 5 个节点，在这个集合中建立 5 个链表，分别代表这 5 个节点，然后将每个节点的所有邻居作为元素插入到链表中。假如 AB 有一条边的权值是 5，我们就在 A 的这个链表中存储节点 B，并记下值为 5 即可；BC 有一条边的权值为 6，我们就在 B 这个链表中存储节点 C，并记下值为 6 即可。



邻接表

小可：嗯，有边就记录，没有边就不记录，这样确实很节省存储空间。

Mr. 王：不过邻接表也不是完美的，当图比较稠密的时候，图中的边就特别的多，链表中的元素也就特别的多。链表上不止有数据域，还有一个指针，相比邻接矩阵，这个指针完全是浪费空间的，它没有存储任何与图有关的内容。所以对于稠密图，邻接表的表现不佳。

综合来看，这两种存储结构是各有优缺点的，不能单纯地说哪一种结构就优于另一种结构。这种道理也是普遍存在的，没有完美的结构，只有最适合的结构。这要看具体的数据规模、结构情况和使用的算法更适合于哪一种结构来进行选择，才能更节省空间或者时间来更好地解决问题。

Mr. 王：关于图有很多的经典算法，比如单源最短路径、最小生成树等。在我们的讨论课中，我会给出这些经典算法的大数据版本。当然，在那之前，我会带你复习其经典版本。

4.3.2　平面图直径

小可：好的，关于图的基本内容我听懂了。

Mr. 王：很好，图能够对很多现实问题进行数学抽象，方便通过计算机的手段进行抽象。而平面图指的就是可以铺在平面上的图，且这个图铺在平面上时仅能在顶点处相交，边与边之间不能相交。我们要求出平面图的直径。

小可：图的直径，就是图中最远的两个点间的最短距离吧。

Mr. 王：是的。在这个问题中，我们已知的是任意两点间的最短路径，要求的是图的直径。你来说说这个问题的输入输出，再来分析一下问题的输入规模。

小可：

输入：有 m 个顶点的平面图，任意两点之间的距离存储在矩阵 $\boldsymbol{D}$ 中，即点 i 到点 j 的距离为 D_{ij}。

输出：最大的 D_{ij} 也就是图的直径。

一共有 m 个顶点，每两个顶点间都有一个最短距离，所以输入数据一共有 m^2 个。

Mr. 王：很好，我们就设 $m^2=n$，同时简化一下这个问题，这个图满足这样的要求，即点与点之间的距离是对称的，而且满足三角不等式。

小可：这就像现实中的双向公路一样，A 地到 B 地的距离和 B 地到 A 地的距离是一致的，三角形的两边之和大于第三边，嗯，这也是很朴素的。不过我觉得这个问题只要扫描一次整个数组，看看哪个值最大不就可以了吗？

Mr. 王：这样的确可以得到正确答案，不过对于 n 来说它是线性时间的，可是我们要求的时间复杂度是亚线性的 $o(n)$。

小可疑惑地说：如果连所有的数据都不能遍历一次，万一没有遍历到的数据就是最大的，那岂不是造成了结果的错误？

Mr. 王：前面我们提到过，规定的时间界限比较苛刻，我们无法在规定的时间内得到精确解，有的时候不得不牺牲一些精度，通过近似算法给出一个近似解，这个近似解“差不多”是我们可以接受的就好。但是非常重要的一点是，我们一定要知道这个近似结果“差多少”。

我们先来看看这个算法：

（1）任意选择（$k \leqslant m$）。

（2）选择使得 D_{kl} 最大的 l。

（3）输出 D_{kl}。

小可思索了一下，说：这个结果还真是差不少啊。

Mr. 王笑着说：那我们就来看看这个结果究竟“差多少”。在这个例子中，我们试着求出算法得出的值和精确值的比是多少。

首先，$D_{ij} \leqslant D_{ik} + D_{kj}$，这是三角不等式的结论。又有 $D_{ik} + D_{kj} \leqslant D_{kl} + D_{kj} = 2D_{kj}$，这是由算法的第 2 步决定的，每个 D_{kx} 都要比 D_{kl} 小。于是，$D_{ij} \leqslant 2D_{kj}$，因而近似比为 2。也就是说，即使在最坏的情况下，也不会小于最优解的 1/2。

那么时间复杂度是多少呢？

小可：输入一共有 n 个距离，而我们只访问了数组中的一行，也就是 m 个数据。又因为 $n=m^2$，所以复杂度是 $O(m)$，也就是$O(\sqrt{n})$。$O(\sqrt{n}) \in o(n)$，它的确是一个亚线性算法。

Mr. 王：很好，我们来总结一下关于近似算法的内容。近似算法是一类主要求解最优化问题的算法，但近似算法给出的不是最优解，而是近似最优解，不过其效率要远远高于求出精确解的算法，这才是提出近似算法的意义所在。

但是我们一定要知道，近似解和最优解究竟相差多少。一般来说，有 3 种衡量办法：**近似比、相对误差**和 $(1+\varepsilon)$ **近似**。

其中最常用的是近似比（Ratio Bound），近似比是这样定义的：

如果近似算法 A 具有近似比 $p(n)$ 的话，那么满足

$$\max\{\frac{C^*}{C}, \frac{C}{C^*}\} \leqslant p(n)$$

其中 n 是输入规模的大小，C 是近似解的代价，C^* 是最优解的代价。

小可：那为什么要取两种比的较大值呢？

Mr. 王：因为我们求解的最优化问题不只包含最大化问题，还有最小化问题。对于最大化问题，C^* 比 C 小；而对于最小化问题，C 比 C^* 要大。所以这样定义才能覆盖两种问题。可以看出，近似比是一个大于 1 的值，而且其越大，说明这个算法越坏，或者说与最优解差得越远。

小可：那如果近似比为 1，得到的就是精确解了吗？

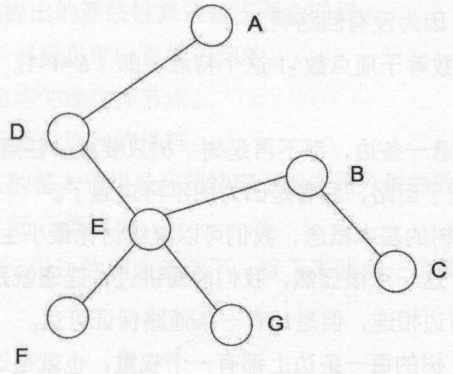
Mr. 王：是的。现在简单介绍一下相对误差。相对误差就是对于任意输入，有 $|C-C^*|/C^*$，其中 C 是近似解的代价，C^* 是最优解的代价；而如果一个近似算法满足 $|C-C^*|/C^* \leqslant \varepsilon(n)$，那么其相对误差界为 $\varepsilon(n)$。

4.3.3 最小生成树

Mr. 王：我们再来讲一个时间亚线性算法——最小生成树问题。这里先简单介绍一下树的概念。

小可：那什么是树呢？

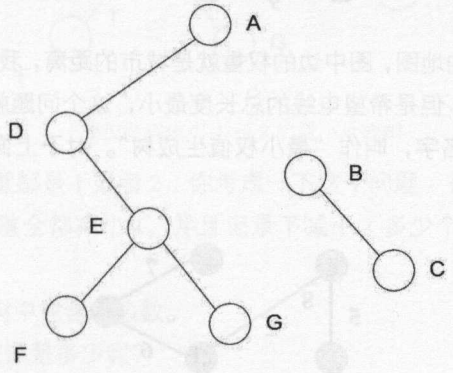
Mr. 王：树的简单定义，就是一个没有回路的连通无向图。树本身也是一个图，但是它的特点是首先要连通；其次不能有回路。



一棵树

小可：哦，这么说倒是形象了很多，自然界中的树的每个部分都是连在一起的，如果断开了就不是这棵树的部分了。

Mr. 王：如果断开了，也就是说，一个无向图满足无回路，却不满足连通，则称作森林。



森林

小可惊讶地说：连通的叫树，那这种不连通的“树”反而叫森林?

Mr. 王笑着说:如果把森林看成断开的树,可能不好理解“森林”这个名字,但是你仔细想想,是不是森林中的每一个连通分量都是一棵树啊?

小可：好像还真是这样，因为其中的每一个连通分量都是没有回路的、连通的无向图!

Mr. 王：所以森林也可以看成是树的集合。

小可：哈哈，这样就更符合它的定义了。

Mr. 王：树在计算机科学中非常常见而且非常重要。树由于有了这个限制，才有了很多有趣的性质。

比如说树中任意两个顶点之间仅存在唯一的一条简单路径。

小可：这个很好理解，因为没有回路啊。

Mr. 王：而且它具有边数等于顶点数 -1 这个特点，即：$E=V-1$。

小可：还真是。

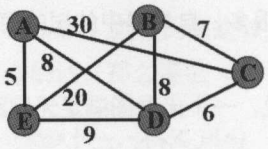
Mr. 王：树只要添加任意一条边，都不再是树；树只要减少任意一条边，也都不再是树。

小可：前者是因为出现了回路，后者是因为图不再连通了。

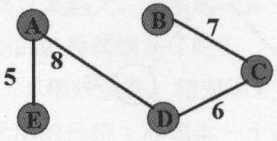
Mr. 王：很好，掌握了树的基本概念，我们可以继续讨论最小生成树的问题了。

最小生成树是连通的，这一点很显然，我们前面讲过，连通就是任意两个顶点之间都是可达的，虽然它们之间未必有边相连，但是却有一条通路保证可达。

在最小生成树问题中，树的每一条边上都有一个权重，也就是这条边的长度。最小生成树问题有一个很好的例子，比如下面的图：



假设这是一张城市间的地图，图中边的权重就是城市的距离，我们要在城市之间架设电线，以保证两个城市之间连通，但是希望电线的总长度最小，这个问题就是最小生成树问题。其实最小生成树有一个更好的名字，叫作“最小权值生成树”。对于上面的这个图来说，求出的最小生成树就是如下这样的：



求它的经典算法有两个，即 Kruskal 和 Prim。下面我简单介绍一下 Kruskal 算法。

（1）首先将所有的边排序。

（2）不断地取出权值最小的边加入最小生成树中，直到所有的顶点都被连通。同时判断：

- 如果最小生成树中出现环路了，就将新来的边移除。
- 否则保留它，继续执行第 2 步。

小可：哦，这样由于不断取出的边都是最短的边，同时还保证了不出现环路，所以它产生的就是最小生成树吧。

Mr. 王：没错，但是它的复杂度是超过线性的，为 $O(m\log n)$，所以这个算法仅在图比较小的时候能较快地得出结论。

小可：我们还是要寻找亚线性算法。

Mr. 王：对，这里我们提出的亚线性算法基于两个前提：

第一，每个顶点的每个邻居是可以直接访问的。

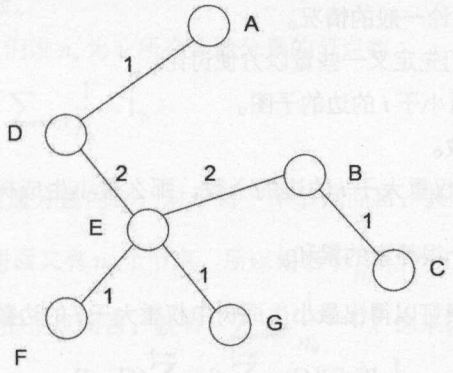
第二，我们可以随机而均匀地选择节点。

这意味着我们可以进行随机均匀的抽样。

我们设计的亚线性算法的基本思想是利用特定子图连通分量的数量估计最小生成树的权重。

小可：这个太抽象了，听不懂啊。

Mr. 王笑了笑，说：这个说法的确太复杂了，接下来我用一个简化的例子解释一下。



一棵仅有权值为 1 和 2 的最小生成树

假设图中所有边的权重都是 1 或者 2，你考虑一下这个问题：在最小生成树中包含的这些边中，如果将所有边的权重全都减小 1，并且记录下减小了多少个 1，记为 S，那么这个量 S 代表了什么？

小可：就是最小生成树中包含的边数。

Mr. 王：那这个边数应该是多少呢？

小可：要把图中的 n 个顶点全部连接起来，3 个顶点至少需要 2 条边，4 个顶点至少需要 3 条边，那么 n 个顶点至少需要 n-1 条边！

Mr. 王：很好，这也恰好是树中顶点数和边数的关系。如果将最小生成树的边数表示成这个式子，那么对于我们做出假设的这个图，最小生成树的权重 $=\#N_1+\#N_2$。

其中的 $\#N_i$ 表示最小生成树中权重至少为 i 的边的数量，那么最小生成树的权重可以表示成什么？

小可：因为最小生成树有 n-1 条边，所以就是 n-1+$\#N_2$

Mr. 王：好，我们现在只把权值为 1 的那些边加进来了，这可能会导致什么？

小可：有很多节点是由边权为 2 的边连接着的，如果只考虑边权为 1 的边，最小生成树就不连通了。

Mr. 王：没错，的确会出现这样的问题。前面我们已经给这些不连通的树起了名字，叫作

连通分量。那么这些连通分量组成最小生成树要用什么来连接呢？需要多少条边呢？

小可：要用权值为 2 的那些边。如果这些连通分量都看作是顶点的话，比如有 m 个顶点，那么就需要 m-1 条边！这和构成连通图是一致的。

Mr. 王：非常好，我们的结果就可以表示成：

n-1+ 权重为 1 的边构成的导出子图的连通分量数 -1

如果这个图中的边权有 1、2、3，那么其做法和上面的也是一样的，只是此时我们要考虑的是 $\#N1$、$\#N2$、$\#N_3$。

Mr. 王：接下来我们讨论一般的情况。

在一般的情况中，我们先定义一些量以方便讨论。

G_i：G 中包含所有权重小于 i 的边的子图。

C_i：G_i 中的连通分量数。

令 β_i 为最小生成树中权重大于 i 的边的个数，那么最小生成树的权重之和就可以表示成 $W_{\mathrm{MST}}(G)=\sum_{i=0}^{w-1}\beta_i$。这是一个很朴素的累和。

由我们前面讨论的结果可以得出最小生成树中权重大于 i 的边数为 C_i-1。

$$W_{\mathrm{MST}}(G)=\sum_{i=0}^{w-1}\beta_i=\sum_{i=0}^{w-1}(C_i-1)$$

其中的常数 1 可以提出来，结果就是：

$$W_{\mathrm{MST}}(G)=\sum_{i=0}^{w-1}(C_i-1)=-w+\sum_{i=0}^{w-1}C_i$$

这里还有一个特殊的 C_0，它的值就是所有的边，也就是 n，我们把它提出来，就可以得到最终的结论：

$$W_{\mathrm{MST}}(G)=n-w+\sum_{i=1}^{w-1}C_i$$

你来说说看，这个过程建立了哪两个量之间的关系？

小可：建立了最小生成树的权重 $W_{\mathrm{MST}}(G)$ 和连通分量的个数 C_i 之间的关系。

Mr. 王：很好，此时问题就转变成了，当拿到一个图之后，如何快速地估计这个图的连通分量的个数。当先来看看基础算法和它存在的问题。我们定义问题为

输入：图 $G=(V, E)$，有 n 个顶点，表示为邻接矩阵，节点最大度为 d。

输出：连通分量的个数。

对于经典算法，简单来说，就是对于每个顶点，我们都要研究其邻居节点，这样它的时间复杂度为 $\Omega(dn)$。

小可：这样它就不是一个亚线性算法了。

Mr. 王：是的，一旦图的顶点很多，计算起来就变得非常费时。我们还是要寻找一种随机化方法来解决这个问题。接下来我给出的这个算法可以满足：

- 用来估计连通分量个数 $\#CC$。
- $\Pr(\#CC \pm \varepsilon n) \geqslant \frac{2}{3}$。
- 同时在时间复杂度上和 n 无关。

Mr. 王：下面我们讨论一下这个算法的核心思想。

设 C 为连通分量的个数。

对于每一个节点 u，我们设 n_u 为 u 所在连通分量的节点数。

对于每一个连通分量：$\sum_{u \in A} \frac{1}{n_u} = 1$。

小可：这是为什么呢?

Mr. 王：因为在一个连通分量内部，对于每一个节点而言，其节点数的倒数都是一样的，均为 $\frac{1}{n_u}$，在这个连通分量里面又有 n_u 个节点，所以相当于 $\frac{1}{n_u} \cdot n_u = 1$。

Mr. 王：那么对于所有的顶点而言，就有：$\sum_{u \in V} \frac{1}{n_u} = C$。你来说说看，这是为什么?

小可：我知道了，因为对于每一个连通分量，都有 $\sum_{u \in A} \frac{1}{n_u} = 1$，只要把所有的连通分量加起来就是包含所有顶点的情况，而在式子的右侧每一个连通分量贡献 1，则所有的连通分量之和自然就是 C 了，即 $\sum_{u \in V} \frac{1}{n_u} = C$。

Mr. 王：很好，接下来我们只要想办法求出 n_u 就可以了。解决它的基本思想还是抽样。问题就是，这个抽样怎么做?

如果 u 所在连通分量的顶点数很少，那么用图搜索算法遍历它就会很容易，只需要很短的时间。

如果 u 所在连通分量的顶点数很多，我们计算出精确的 n_u 就会变得非常困难，同时 $\frac{1}{n_u}$ 会变得很小，对整个求和的贡献也就变得很小。所以当用图搜索算法遍历一个连通分量产生困难，我们不妨忽略它。

设对 n_u 的估计值为 $\hat{n}_u = \min(n_u, \frac{2}{\varepsilon})$，这个式子很好地解释了我们之前讨论的情况。即：

- 当节点数小于 $\frac{2}{\varepsilon}$ 时，我们认为它可以正常完成遍历，显然有 $\hat{n}_u = n_u$。
- 当节点数大于 $\frac{2}{\varepsilon}$ 时，我们就认为节点已经多到难以遍历了，就将其估计值定为阈值 $\frac{2}{\varepsilon}$，即 $\hat{n}_u = \frac{2}{\varepsilon}$，则 $\frac{1}{\hat{n}_u} = \frac{\varepsilon}{2}$。

小可：王老师，我们为什么要取这么奇怪的数呢？

Mr. 王笑了笑，说：后面我们还需要这个值来凑$\frac{2}{3}$。接下来你想想，当n_u比较大时，这个估计误差是多少？

小可：估计误差是$\frac{1}{\hat{n}_u}-\frac{1}{n_u}$。$\hat{n}_u$小于实际值，而且$\frac{1}{\hat{n}_u}$减去一个正数小于其本身，即$0<\frac{1}{\hat{n}_u}-\frac{1}{n_u}<\frac{\varepsilon}{2}$。综合起来就是$0<\frac{1}{\hat{n}_u}-\frac{1}{n_u}<\frac{\varepsilon}{2}$。

Mr. 王：不错，我们来整理一下对C的估计。

我们将对C的估计表示为$\hat{C}=\sum\frac{1}{\hat{n}_u}$，那么它的误差就是：

$$|\hat{C}-C|=|\sum(\frac{1}{\hat{n}_u}-\frac{1}{n_u})|\leqslant\frac{\varepsilon n}{2}$$

小可：至此，我们成功地证明了误差是有界的。那么具体的算法又是怎样的呢？

Mr. 王：接下来我们给出具体的算法：

- $CC(G,d,\varepsilon)$

1 for i = 1 to s = $\theta(\frac{1}{\varepsilon^2})$ do

2 随机选择点 u

3 从 u 开始 BFS（广度优先搜索），将访问到的顶点存点到排序队列 L 中，访问完连通分量或 L=2/ε 时停止，$\hat{n}_u=|L|$

4 N=N=+ $\hat{n}_u$

5 返回$\tilde{C}$ =s/N×n

小可：第 1 行我就没有看懂，$s=\theta(\frac{1}{\varepsilon^2})$是怎么回事呢？

Mr. 王：这个表示抽样要与$\frac{1}{\varepsilon^2}$同阶，但是具体怎么抽取，接下来我们还会进一步讨论。现在你只要知道这一步是进行抽样即可。

第 2 行很简单，我们随机选取抽样中的一个点。

在第 3 行中，我们用选取的这个点，在连通分量内执行图搜索算法，将访问到的顶点都放在排序队列L里面，如果不能将队列填满到$\frac{2}{\varepsilon}$，我们就取队列内元素的个数n_u；否则，一旦L的长度为$\frac{2}{\varepsilon}$，我们就将n_u的估计值取$\frac{2}{\varepsilon}$。

第 4 行，将估计值进行累加。

第 5 行，根据前面的推导，给出一个对连通分量个数的估计，$\hat{C}=\frac{s}{N}\cdot n$。这里体现的就是一个样本估计总体的思想。

我们来看看整个算法的时间复杂度：$O(\frac{d}{\varepsilon^3}\log\frac{1}{\varepsilon})$。

小可：这个值可是够复杂的。

Mr. 王：你别看它的形式很复杂，但从式子中不难看出，复杂度与图的大小是无关的，说明这是一个很好的亚线性算法。我来仔细解释一下这个复杂度的构成。

首先，整个外侧的循环是与$\frac{1}{\varepsilon^2}$同阶的，而在每一次循环中，最多就处理$\frac{2}{\varepsilon}$个节点；其次，当我们把节点放进 L 中时，由于这个 L 是一个排序队列，将一个新的元素插入一个排序队列中的时间效率为 $O(\log\frac{1}{\varepsilon})$，所以相乘后的结果就是$O(\frac{d}{\varepsilon^3}\log\frac{1}{\varepsilon})$。

Mr. 王：好了，到现在为止，我们已经设计和分析了对连通分量进行估计的近似算法：

1 for i=1 to w-1 do

2 $\tilde{C}_i = CC(G_i, d, \frac{\varepsilon}{w})$

3 return $\tilde{w}_{MST} = n - w + \sum_{I=1}^{W-1}\tilde{C}_i$

其中，*CC* 是对连通分量数量的估计，是我们前面已经完成建立的算法。经过前面的分析，这个算法就已经很好理解了。

4.4　时间亚线性判定算法

4.4.1　全 0 数组的判定

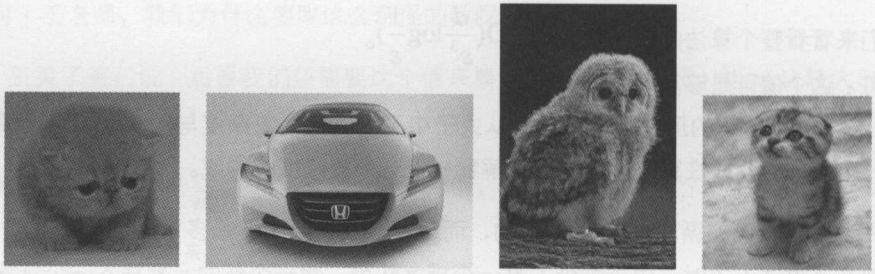
Mr. 王：接下来我们讲一类时间亚线性判定算法，先来举个例子吧。假设有一个数组 A，其中包含 0 和 1，我们需要判定数组里面的元素是否全是 0，如果全是 0，则输出“是”；否则输出“否”。依然要求时间复杂度为 $o(n)$。

小可：$o(n)$ 意味着我们还是无法访问全部数据，可是我觉得判定问题和最优化问题是不一样的。在最优化问题中，虽然得不到最优解，但是可以返回一个近似解，只要知道这个近似解和最优解差多少就可以了。但是这种判定问题只回答一个“是”或者“否”，如果还是差不多的话，岂不是答错了吗？

Mr. 王：对于判定问题，则换了一种近似的方法，我们的回答是输入满足某种性质，或者远非满足某种性质。

小可：远非满足？

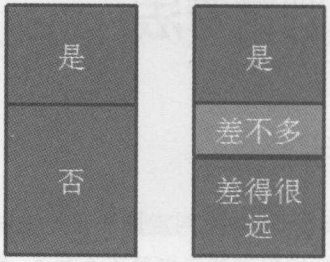
Mr. 王从抽屉里拿出了几张图片，一张画着猫，一张画着汽车，一张画着猫头鹰。



“猫”和“远非猫”

Mr. 王：你看看这几张图片，假设我们的算法是要识别某张图片里面是不是画着猫。第一张图片没什么说的，它一定满足有猫这个性质；第二张图片里面画着汽车，这就是远非满足有猫这个性质；而第三张图片就比较复杂了，里面画着猫头鹰，这需要算法仔细地进行判断，而这种判断需要消耗的时间就会比较长，如果时间条件要求比较苛刻，我们就真可能将其判断为有猫了。

对于判定问题的严格精确解，我们能给出严格的是或者否；而对于判定问题的近似算法，只要给出“是”和“差得很远”这两种情况就可以了；对于那种“差不多”的情况我们不做区分。在上面的例子中，近似算法只要能够找出猫就可以了，对于猫头鹰，我们可以不做区分。



小可：嗯，但是怎么定义这个“差得很远”和“差不多”呢？

Mr. 王：这里提出一个概念叫作“**ε- 远离**”。对于输入字符串 x，如果从 x 到字符串集合 L 中任意字符串的汉明距离至少为 $\varepsilon|x|$，则 x 是 ε- 远离 L 的。

Mr. 王：那么你来想想，这个问题应该怎么解决呢？

小可：既然处理不了所有的元素，那么只要挑出一部分来看看它们是不是 0 就可以了，这就是抽样！

Mr. 王：很好，就是利用抽样算法，具体算法是这样的：

（1）在 A 中随机独立抽取 $s=2/\varepsilon$ 个位置上的元素。

（2）检查抽样，若不包含 1，则输出“是”；若包含 1，则输出“否”。

小可：嗯，抽取这个特殊的数量有什么含义呢？

Mr. 王：我们就来分析一下这个算法的精确性。当数组中包含 1 却没有被抽取出来时，算法就会出现误报，这种本来应该返回否而算法返回是的情况，我们称作**假阳性**。如果 A 是 ε-远离的，那么误报的概率就是：

$$\Pr(\text{抽样中没有 1}) \leqslant (1-\varepsilon)^s \approx e^{-\varepsilon s} = e^{-2} < \frac{1}{3}$$

这是为什么呢？我们每次抽取到 1 的概率是 ε，那么抽取到 0 的概率就是 $1-\varepsilon$，最后抽取 s 次每次都抽取到 0 的概率就是 $(1-\varepsilon)^s$，后面的部分是概率统计中的近似结论，了解即可。

小可：而且这个算法的运行时间还与数组的长度无关，仅跟抽样次数 s 有关，时间复杂度就是 $O(s)$，而 s 是远小于 A 的长度的，所以它是一个时间亚线性算法。

Mr. 王笑着说：很好，分析得很对。下面再给出一个证据引理：

如果一次测试以大于等于 p 的概率获得一个证据，那么 $s=2/p$ 轮测试得到证据的概率大于等于 2/3。

对于判定问题 L，其查询复杂性为 $q(n)$ 和近似参数 ε 的性质测试算法是一个随机算法，其满足对于给定 L 的是一个实例 x，最多进行 $q(|x|)$ 次查询，并且满足下述性质：

- 如果 x 在 L 中，该算法以最小 2/3 的概率返回“是”。
- 如果 x 是 ε-远离 L 的，该算法以最小 2/3 的概率返回“否”。

4.4.2　数组有序的判定

Mr. 王：这里我们再讲一个亚线性时间的判定问题——数组有序的判定问题。你来说一下问题定义，并想一想这个问题的精确解。

小可：

输入：n 个数的数组，$x_1, x_2, \cdots, x_n$。

输出：如果数组有序则返回“是”，否则返回“否”。

如果是求精确解的话，需要逐个元素与后面的元素进行比较，一旦发现有逆序的情况，返回否就可以了。可是这样做的时间复杂度是 $\Omega(n)$，当数据有很多的时候，这个算法是不适用的。

Mr. 王：很好，现在你分析问题已经很成熟了。这里同样要提出一个近似判定算法。我们要确定的是，这个数组是有序的，还是 ε-远离有序的。

小可：在这个问题中，ε-远离有序是怎么定义的呢？

Mr. 王：在此问题中，如果删除数组中多于 εn 个的元素会使数组有序，我们就称这个数组为 ε-远离有序的。这意味着问题变成了，数组是有序的，还是要删除数组中多于 εn 个的元素才能使之有序的判定问题。

小可：既然不能访问整个数组中的元素，那么我们还是以采样的方式来进行吗？

Mr. 王：的确要通过采样的方式，但是重要的是，对于这个问题我们怎么采样。

这里要补充一个预备知识，叫作二分查找算法，这是一个非常经典的算法。

利用二分查找法就是希望能在一个递增的序列 $S<x_1,x_2,x_3,\cdots,x_n>$ 中查找某一个值 x，具体的做法是这样的：首先找到 S 的中位数 mid，然后与 x 进行比较，如果 x=mid，直接返回位置就可以了；如果 x>mid，就把新的查找区间定为 (mid,x_n)，否则定为 (x_1,mid)；依此类推，直到查找到 x 的位置。

二分查找的时间复杂度是对数时间的，也就是 O(log$n$)。这里我们先对其进行简单的解释，后面会详细地根据有根树的性质讨论它的复杂度问题。

算法的第 1 步，我们面对的是整个数据序列，所选择的数字是比中位数小还是比中位数大，这样相当于将整个序列划分为两部分，一部分是比中位数小的一半，另一部分是比中位数大的一半。

第 2 步，数据集合中只剩下了我们要访问的一半，再从这一半中找到一半。

好了，我们回到数组有序判定这个问题上，来看看下面这个算法：

```
1 for k=1 to 2/ε do
2   随机选择数组中第 i 个元素 x_i
3   用 x_i 在数组中做二分查找
4   if 发现 i<j，但是 xi>xj then  // 碰到了“坏”索引
5       return false
6 return true
```

算法的第 4 行表示，一旦发现两个数前面的比后面的小，就说明这个数组是无序的，我们称之为“坏”索引。

这个算法的时间复杂度为 $O(\frac{1}{\varepsilon}\log n)$，因为外面的循环执行了$\frac{2}{\varepsilon}$次，2 是常数 c 就可以忽略了。至于后面的 log$n$，是因为二分查找的时间复杂度是 log$n$。log$n$ 的阶是要比 n 低的，即 log$n$=o(n)，说明这是一个亚线性算法。

小可：这个算法的准确度如何呢？

Mr. 王：如果输入的数组是有序的，那么一定会返回“是”。我们要证明的就是，对于一个 ε-远离的数组，准确率可以达到$\frac{2}{3}$。

小可：我想起了全 0 数组的判定问题。

Mr. 王：差不多。首先回忆一下我们前面讲过的证据引理。我们来证明这么一个问题：如果输入 ε- 远离有序，则存在大于 εn 个的“坏”索引，也就是前面算法中提到的逆序。

证明：一个命题和它的逆否命题同真假，我们不妨来证明它的逆否命题，就是如果“坏”索引的个数小于 εn，则其存在一个长度大于 εn 的单调递增的子序列。我们可以考虑，如果“坏”索引都被剔除的话，留下的就是一个单调递增的子序列了。

对于任意“好”索引 i 和 j，有 $x_i<x_j$。

令 k 是在二分搜索中将 x_i 和 x_j 分开的最近顶点，也就是对于整个数组建立一个二分搜索树，在二分搜索树中 x_i 和 x_j 的最近公共祖先，则 $i<k<j$，因为 i 和 j 都是“好”索引，那么 $x_i<x_k<x_j$。

现在我们回到证明其准确率的问题上。

我们要证明当输入数列 ε- 远离有序时，算法返回 false 的概率大于 2/3。

证明：往证算法返回 true 的概率小于 1/3。

我们已经证明，如果输入 ε- 远离有序，则存在大于 εn 个的“坏”索引，即数组中“坏”索引的概率大于 ε。

当数组中“坏”索引的概率大于 ε 时，我们经过了 2/ε 次选择，每次选择是“好”索引的概率是 1-ε，2/ε 都是“好”索引的概率就是 $(1-\varepsilon)^{2/\varepsilon}<e^{-2}<\frac{1}{3}$。这样算法的精确性也就得到了证明。

小可：嗯，这和全 0 数组判定的证明挺像的。

Mr. 王：好了，亚线性算法就讲解到这里，下次课我们来讨论磁盘算法。

第5章 价钱与性能的平衡——磁盘算法

5.1 磁盘算法概述

Mr. 王：现在我们谈谈磁盘算法的问题。根据你的了解，跟我说说计算机中都采用了哪些种类的存储器？

小可：这个我还是略知一二的。计算机中有很多用来存储数据的存储器，比如寄存器、缓存（Cache）、内存和硬盘等。

Mr. 王：这些存储结构都有什么特点呢？

小可：寄存器、缓存和内存都是需要依靠电来维持其所存储的数据的，而磁盘可以在断电的情况下保存数据。数据是存储在磁性介质上的。

Mr. 王：它们的速度、容量和价格又如何呢？

小可：它们的容量是依次变大的，但访问速度却是越来越慢的，而且越快的存储器相对也就越昂贵。

Mr. 王：是的，计算机中存储器的价格衡量标准是单位存储空间的价格。寄存器在计算机中的存储空间是比较小的，现代计算机中的寄存器一般都直接集成在 CPU 内部，但寄存器的存取速度是最快的，往往可以用来存储 CPU 运行的中间结果。Cache 的价格比寄存器稍便宜，存储空间也比寄存器大，它的存取速度比寄存器稍慢一些，它往往处在内存和 CPU 中间，用

来缓存数据，使得当 CPU 需要用到一些数据时，可以快速地找它们，而不是到内存中去寻找。下一级别的存储器是内存，内存也叫主存，存储空间相对较大，它的存取速度比 Cache 慢，但比各种外存设备快得多，一般用于在程序执行时存储程序和必要的数据，我们在运行程序时，也要将程序从磁盘加载到内存中。价格最便宜而存取速度最慢的设备叫作辅存，计算机中最典型的辅存就是磁盘，当然常见的还有光盘等。它们适合存储大量的数据，而且这些数据存储在磁性介质或者光介质上，并不依赖于电力，在系统断电之后数据也不会消失。

从价格、存取速度和空间的角度来看，设置多级存储结构还是非常有必要的，这能够让我们在最节省钱的情况下，达到最好的存储和访问效果。

小可：我知道多数程序都是在内存中运行的，那我们为什么还需要磁盘算法呢？

Mr. 王：其实这个很显然啊，内存的容量是有限的，我们在处理很多问题时，需要比现有内存更大的空间来存储数据，自然也就需要磁盘这样的大容量存储介质来帮忙了。相对于内存来讲，像磁盘、磁带这样的存储介质一般称作外存，所以磁盘算法也叫外存算法。

小可：那么把数据存放在内存中和存放在磁盘中，我们在进行处理时有什么区别呢？

Mr. 王：我们知道，内存是可以进行随机访问的。也就是说，给出一个内存地址，我们就可以很快地把数据找出来。硬盘可就不一样了，它是由一层层的盘片组成的，磁盘上有一个读写头，通过读写头的旋转来找到盘片上的不同位置。如果要在硬盘上找两个数据，在找到第一个数据之后去寻找第二个数据时，需要旋转读写头，如果这两个数据的距离比较远的话，读写头需要旋转的距离也就很远。这导致硬盘的读写速度是很慢的，其访问速度比内存要慢上 106 倍。

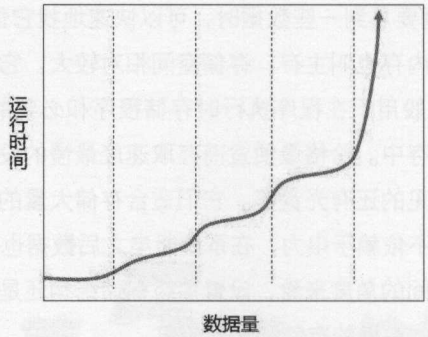
小可：那的确是够慢的了。

Mr. 王：想想看，这个问题可以怎么解决？

小可：访问开销比较大的原因就是磁盘读写头的旋转消耗了时间，如果需要连续读取的两个数据离得很近或者是相邻的，那么旋转读写头的时间就很短，可以很快速地把数据取出来，额外的开销也就会变得非常小，因此我们只要把数据都连起来存储就好了。

Mr. 王：不错，所以硬盘尽可能将连续读取的数据放在一起，用大规模连续的数据传输来平摊消耗。因此，硬盘的一个重要特点就是它以块为单位进行访问，一般这个块的大小是 8 ~ 16KB，以保证高效地进行数据存取。

在以往的计算模型中，我们关注的只有 CPU 和内存，程序和各种数据存储在内存中，CPU 调取这些数据进行处理，而当数据量比较大时，或者是操作系统到硬盘中读取内容时，就不得不考虑磁盘输入输出（磁盘 I/O）带来的额外开销了，这也是我们要来讨论磁盘算法的意义所在。



数据量和程序运行时间的关系

上面的图说明了数据量和程序运行时间的关系。不难看出，当数据量大到一定程度时，运行时间会产生一个阶跃。这个阶跃就出现在数据量突破了内存的容量时，我们不得不将数据存储在磁盘中，然后不断地将需要用到的数据加载进内存中，再进行进一步的处理。另外，当算法不够好时，虚拟内存会不断地访问磁盘，不停地产生掉页错误，最终导致算法的运行开销大大增加。

小可：看来设计好的磁盘算法还真的很有必要啊。请老师给我举几个磁盘算法的例子吧。

Mr. 王：先别着急，我们要先建立一个磁盘模型，因为之前的算法都是运行在内存中的。

我们用圆柱体来代表磁盘，矩形框就是内存。内存和磁盘之间的 I/O（输入输出）交互是以块为单位进行的。我们先来设几个基本项：N= # 问题实例数据项个数

B= # 每个磁盘块中的数据项个数

M= # 内存能容纳的数据项个数

T= # 输出数据项个数

I/O= # 内存和磁盘之间移动的块数

简单解释一下这几个量。N 就是我们在分析算法中使用的 n，用它来表示算法需要处理的数据规模。B 是每个磁盘块可以容纳的数据项个数，前面我们提到，数据在磁盘上是以块为单位存储的，而对于每个问题而言，一个数据的大小也是不同的，所以用 B 来表示一个磁盘块可以容纳多少单位数据。M 表示内存中可以容纳的数据项个数，也就是我们为这个程序存储分配的内存空间可以容纳多少个数据。T 是输出数据项的个数，在我们的分析里很少涉及这个概念。我们把内存和外存交换一个磁盘块定义为一次 I/O，这个 I/O 量表示的就是内存和磁盘之间进行了多少次磁盘块交换。

另外，在这里要给出一个非常重要的假设：$M>B^2$。这就是说，内存的大小要比磁盘块大小的平方大。如果没有这个假设的话，很多磁盘算法的设计与分析会变得异常复杂，而且这个条件在现代的计算机系统中是非常容易满足的。所以我们就基于这个假设，来展开对磁盘算法的研究。

这里还要介绍一个内容，就是根据计算复杂性理论，我们要知道在内存和外存中各种基本算法的时间下界。

	内存算法	外存算法
浏览：	N	$\frac{N}{B}$
排序：	$N\log N$	$\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B}$
置换：	N	$\min\{N, \frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B}\}$
查找：	$\log_2 N$	$\log_B N$

小可：为什么内存算法和磁盘算法会产生这么大的区别呢？而且为什么硬盘算法的界限反而变小了呢？

Mr. 王：我们就拿浏览来解释一下吧。如果有 N 个数据，在内存中只要一个个地读取就可以了。而对于磁盘中的数据，我们要先将数据从磁盘读入内存中，而且每次只能读一个磁盘块，所以当一个磁盘块包含 B 个数据项时，就需要进行$\frac{N}{B}$次 I/O。

注意，我们在讨论磁盘算法的时间界限时，是以 I/O 次数为单位进行计算的，而不是以对数据进行一次操作为单位的。简单来说，我们关注的仅仅包括访问磁盘的次数，时间界限和复杂度计算衡量的也就是访问磁盘的 I/O 量。这是由于相比访问磁盘的开销，访问内存的开销已经可以忽略不计了。我们只要能够很好地控制 I/O 的次数，就可以很好地降低磁盘算法的运行复杂度。

小可：原来是这样啊。

Mr. 王：此时，我们对一些概念的定义也就发生了变化，比如在磁盘算法中，线性算法指的是 $O(\frac{N}{B})$ 的算法。

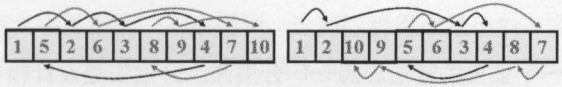
小可：嗯，就以浏览为例，内存算法中的线性算法 $O(N)$ 对应的就是磁盘算法中的 $O(\frac{N}{B})$。

Mr. 王：另外，磁盘块的大小 B 是一个非常重要的量，它对 I/O 复杂度的影响是非常大的。一般来说，$\frac{N}{B} < \frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B} << N$。另外，我们不能用搜索树对排序进行优化，因为不能保证搜索树在磁盘中是连续存储的，如果不是连续存储的，那么开销依然是很大的，达不到优化的目的。

Mr. 王：在前面的式子中，$\frac{N}{B} << N$这一点是非常重要的，这里先举一个最简单的例子来说明这个问题。就以遍历链表（链表排序）为例，数组大小 $N=10$ 个（元素），磁盘块大小 $B=2$ 个（元素），内存大小 $M=4$ 个（元素）。

小可：这就是说，内存中能装下两个磁盘块。

Mr. 王：是的，看一下这两个图：



算法 1：N=10 I/O　　算法 2：N/B=5 I/O

我们注意到，在两个图中，数据都是按照数字编号的逻辑次序进行访问的，但它们存放在磁盘中的物理顺序却是不一样的。在左图中，我们首先访问数据 1，需要从磁盘中加载 [1][5] 这个磁盘块进入内存，然后访问数据 2，加载 [2][6] 这个磁盘块进入内存。访问数据 3 时，内存不足，加载 [3][8] 这个磁盘块，并换出 [1][5]，依此类推。不难发现，在这种磁盘存储的情况下，我们每访问一个数据节点，就需要换入换出一个磁盘块，进行的数据访问次数和 N 的数量级相同，本例中共进行了 10 次磁盘 I/O。

而在右图中，数据节点存储的连续性相对较好，访问数据 1 时，加载 [1][2] 块，访问数据 2 时，不需要加载新的磁盘块，直接访问内存中的 [2] 即可。[3][4] 也在一个磁盘块中，同样只需要进行 1 次磁盘 I/O 就可以访问 3 和 4 两个数据节点。此时，我们是以磁盘块为单位进行磁盘 I/O 的，只需要 N/B 的数量级，本例中共进行了 5 次磁盘 I/O。

小可：单单这么少的数据，N 和 $\frac{N}{B}$ 就差了一半啊。

Mr. 王：在实际情况中，这两个数字的差距更为惊人。

例如：在某个磁盘中，$N=256\times10^6$，B=8000，如果访问磁盘的时间为 1ms 的话，N 次 I/O 需要 256×10^3s = 4266 min = 71 hr，而 N/B 次 I/O 只需要 256/8s= 32s。

小可吃惊地说：71hr 和 32s，竟然可以相差如此之多。

Mr. 王点点头，说：这说明 $O(\frac{N}{B})$ 和 $O(N)$ 不是一个数量级的量，在学习磁盘算法的过程中一定要非常注意，$O(N)$ 对于磁盘来说不是线性算法，$O(N)$ 的算法在实际运行的过程中会产生大量的磁盘 I/O，效率是非常低的。

5.2 外排序

Mr. 王：接下来我们看一看在磁盘算法中一个比较典型的例子——外排序。

小可：那什么又是外排序呢?

Mr. 王：外排序是相对内排序而言的，当要排序的数据量无法被全部装进内存时，我们就需要用到外排序，此时有大量的数据被存在硬盘里，无法直接进行操作，必须先以块为单位读进内存中。

为了更好地理解大数据中的归并排序，我们先从内存中的归并排序说起。该算法被称为“归并排序”或者“多路归并排序”，其基本思想就是，先将整个数组划分为多组，保证每一组内是有序的，然后相邻的两组之间进行“归并”，使得产生的更大的组也是有序的，直到组的大小等于数组的大小。

归并排序体现了一个重要的算法思想——分治法。分治法是设计很多算法时一种有效的策略。对于一个比较大、复杂的问题，我们很难一下子将其解决，这时就尝试采用将大的问题逐

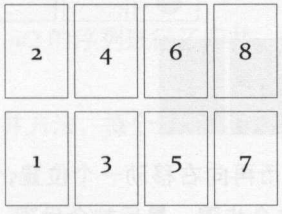
渐划分为小的问题，这些小的问题叫作子问题，对于子问题，求解起来往往会变得容易一些。当然，我们还可以对子问题进行进一步的划分，划分成更小的子问题，在很多问题中，当子问题被划分得足够小时，就可以使用很简单的方法解决或者直接得到解决了。当这些子问题有效地得到解决时，我们就可以按照子问题和原问题的关系逐步将子问题的解进行合并，从而得到原来问题的解。

小可：这也符合我们日常生活中处理问题的思路。对于一个比较棘手的问题，我们会尝试将其分成多个较小的部分，再逐个击破，最后把各部分的解决方案拼到一起，就得到了原来那个大问题的解。

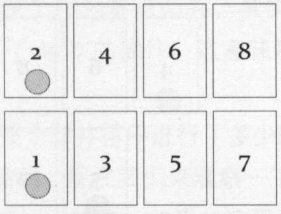
Mr. 王：归并排序其实就是分治法的一种典型体现，在二路归并排序中，首先要将整个乱序的序列划分为两路，然后分别对两路进行一个归并排序。注意，这个过程会递归地进行下去。也就是说，对划分出来的两路继续执行划分，变成四个部分，再从四个部分进行划分，变成八个部分，依此类推。最后直到整个分成了只含有一个元素的序列时，它显然就已经排好序了，我们再把这些排好序的序列进行逐级合并，合并成长度为 2,4,8,16 的序列。

小可：其实划分这个步骤非常容易做，关键就在于合并这个步骤怎么解决，怎么把两路已经排好序的序列合并到一起，成为一个仍然有序的序列呢？

Mr. 王拿出两组扑克牌放在桌面上，说：想一想，假设我们有两个有序的数字卡牌序列，分别是 2468 和 1357，我们要如何将其变成一组有序的数列，即 12345678 呢？

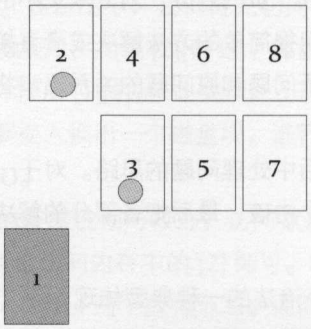


在归并排序的合并中，我们可以用两个硬币来模拟移动的指针。首先，我们把两个指针分别放在两个序列的第一张牌上，由于两路都是有序的，所以这两张牌一定都是两路中最小的。

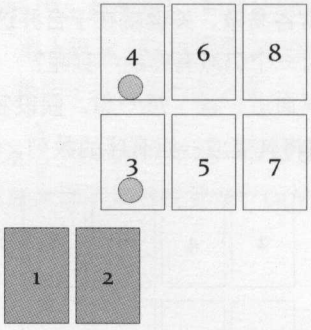


但这两张牌也是有大小关系的，不难想到，这两张牌里面最小的一定是所有 8 张牌里面最小的。根据排序的问题定义，按照从小到大的顺序排列，所以第一次我们希望能够找出序列中最小的那个数。于是，我们比较两个硬币所在的扑克牌，发现 1 比 2 小，所以取出 1，放在外面，

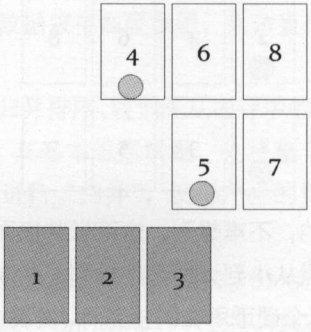
然后将硬币向右移动一个位置，放在 3 上面。现在我们要找出 8 张牌中第二小的。



小可：从 8 张牌中拿出了最小的，那么剩下的 7 张牌中最小的就是第二小的，所以我们只要找到剩下的 7 张牌中最小的就可以了。方法还是和前面的一样，因为现在的两个硬币依然在两组数中最小的两个数上，只要比较它们的大小就可以了。一个是 2，一个是 3，所以取出 2。



Mr. 王：很好，但别忘了将硬币再向右移动一个位置，此时在 3 和 4 上面。我们取出 3，再将硬币向右移动。不断地进行这个步骤，最后就会发现，我们取出扑克牌的顺序刚好就是 12345678。这样就非常有效地将两个大小为 4 的序列合成一个大小为 8 的序列，而同时满足了这个大小为 8 的序列仍然有序这一要求。

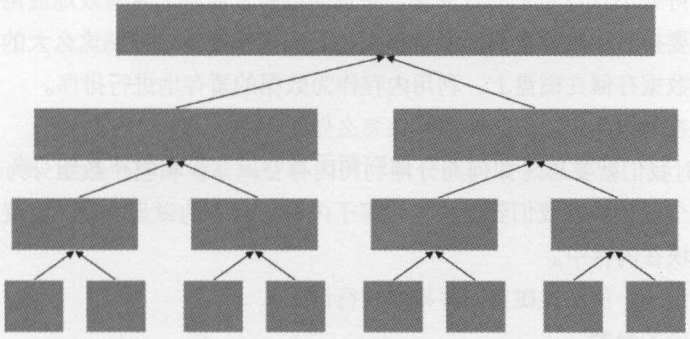


小可：那从 1 到 2，2 到 4，4 到 8，8 到 16 都可以用这个方法进行操作实现，所以对于任意的序列长度 n，我们都可以用这个方法进行排序了。嗯，内存中的归并排序方法我基本上搞懂了。

Mr. 王：别急，我们先来看看这个算法的时间复杂度如何。

小可：这个可有点复杂，要怎么分析呢?

Mr. 王：其实要求解这类问题的时间复杂度，需要用到算法设计分析理论中的一个重要定理，叫作 Master 定理，这是一个可以求解递归式的复杂度的定理。不过，今天我们尝试用一种比较直观的方法进行分析。首先看一下这个图：



模拟归并排序的过程

在最后一轮中，将两个长度为 $n/2$ 的序列进行了归并，合并成长度为 n 的序列。每个元素被访问了几次?

小可：根据我们前面说过的合并方法，每个元素实际上都被访问了常数次，因为就是做了一个简单的比较，所以应该是 cn 次。

Mr. 王：那么倒数第二轮呢?

小可：在倒数第二轮中，有 4 个序列被合并成 2 个，其中一个合并过程有 $n/2$ 个元素参与，所以应该是 $cn/2$，这样的过程应该有两个，所以依然是 cn 次。

Mr. 王：由此发现，不管有多少个序列参与归并，其实在每一轮的归并中，都是访问了所有的元素常数次，每一轮都进行了 cn 次的操作。这意味着归并排序中每一轮的复杂度都是 $O(n)$。

小可：这个我懂了，只要知道整个归并排序进行了多少轮，再乘以 $O(n)$ 就可以了。

Mr. 王：很好，思路是非常正确的。现在我们来观察一下合并这个过程。

Mr. 王指着那个模拟合并过程的图，说道：如果将这里的每一个序列都看作一个节点，将图中的箭头线看作边，那么这是一个什么特殊的图?

小可恍然大悟，说：这是一棵树!

Mr. 王：不难看出，树的每两层之间的边，就代表了进行一次归并。也就是说，树的高度 -1

就是进行归并的次数。

小可：没错，所以我们只要知道树的高度是多少就行了！

Mr. 王：好，这是一棵满二叉树，它有 n 个叶子节点，这课树的高度是多少呢？这里我们先记一个结论，有 n 个叶子节点的满二叉树有 $2n-1$ 个叶子节点，满足高度为 $O(\log n)$。在后面的课程中我们会进一步介绍这个问题。

Mr. 王：综合起来，归并排序的时间复杂度就是 $O(\log n) \cdot O(n)=O(n\log n)$。前面我们提过，基于比较的排序算法，它的时间复杂度下界是 $O(n\log n)$。这说明它已经是一个非常高效的算法了。

归并排序的好处不仅仅是它的效率高，更重要的是它能够非常有效地应用在外排序中。在现实生活中，需要排序的数据量有时候是很大的，当内存中无法容纳这么大的数据量时，我们就要尝试将这些数据存储在磁盘上，利用内存作为数据的暂存地进行排序。

小可：那么在外排序中，归并排序又该怎么做呢？

Mr. 王：此时我们就要思考如何充分地利用内存空间了。将整个数组分为多少路，要考虑内存能装下多少个磁盘块，我们要分的路数等于内存能装下的磁盘块数。也就是说，一定要保证每一路都有一块在内存中。

接下来我们用一个例子对磁盘归并排序进行说明。

先来约定讨论的参数：

N=24，M=8，B=2。

小可：嗯，一共有 24 个数据项，内存能装下 8 个数据项，一个磁盘块包含 2 个数据项。也就是说，内存可以装下 4 个磁盘块。

我们就以 1 ~ 24 这组数字为例吧。其初始状态为：

24 1 23 19 20 5 18 16 4 7 8 9

10 15 17 14 3 2 6 11 12 13 22 21

首先，我们尽可能把内存装满。

<u>24 1</u> <u>23 19</u> <u>20 5</u> <u>18 16</u> 4 7 8 9

10 15 17 14 3 2 6 11 12 13 22 21

每一条下画线代表一个磁盘块大小，用下画线表示它们被装进了内存。

然后，我们对已经放进内存中的这些值进行排序。

1 5 16 18 19 20 23 24

同理，我们也对放入内存中的下两路数字进行排序。

4 7 8 9 10 14 15 17

2 3 6 11 12 13 21 22

小可：现在已经保证了每一路内部是有序的，但是如何借助内存对它们进行归并呢？

Mr. 王：内存只能容下 4 个磁盘块，三路刚好保证每一路可以拿出一个磁盘块放入内存中，剩下的一块我们用来做 I/O 的缓冲区。接下来，我们将每一路的首块放入磁盘中。

1 5

4 7

2 3

[] []

对这几个值进行归并。

较小的 1 和 2 先被放入缓冲区里，同时清空它们在内存中的位置。

[] 5

4 7

[] 3

1 2

此时缓冲区满，将其写回到磁盘里，然后清空缓冲区。

[] 5

4 7

[] 3

[] []

接下来，我们将较小的 3 放入缓冲区中。此时，来自第三路的磁盘块被彻底清空，所以我们从外面调入一个来自第三路的磁盘块。

[] 5

4 7

[] []

3 []

[] 5

4 7

6 11

3 []

接下来，我们将 4 放入缓冲区中，缓冲区满，写回外存。

依此类推，不断地执行这个过程，就可以最终实现外排序了。

小可：那这个外排序算法的 I/O 复杂性又如何呢?

Mr. 王：好，接下来我们就以 M、N、B 三个值为参数，对其 I/O 复杂性进行推导。首先考虑：对整个数据集合进行一次浏览或者说扫描需要多少次 I/O？将内存用数据填满又需要多少

次 I/O？

小可：扫描对于磁盘而言是 I/O 线性的，所以是 N/B。将内存填满的 I/O 次数，也就是内存可以容纳的磁盘块数目，应该是 M/B。

Mr. 王：回想一下，在第一轮归并排序中，我们都做了什么？进行了多少次 I/O？

小可：第一轮，我们把一部分磁盘块装入内存，在内存中排序，然后再把一部分装入内存，在内存中排序……不断地执行，直到所有的数据都进入一次内存！

Mr. 王：所以呢？

小可：相当于对数据执行了一次扫描，所以应该是 N/B 次 I/O！

Mr. 王：很好。那么再想一想，在第二轮的执行过程中，又进行了多少次 I/O 呢？

小可：第二轮，开始执行归并时，我们每一次都将每一路的一个磁盘块放进磁盘中，在不断的归并过程中，其实也是每一个磁盘块都放进一次内存，所以也是 N/B 次 I/O。

Mr. 王：这意味着每一轮都会执行 $\theta(\frac{N}{B})$ 次 I/O。加上一个 θ，是因为我们前面的计算只把输入内存考虑在内了，并没有计算每轮一次的缓冲区满输出到内存，所以使用 $\theta(\frac{N}{B})$ 这个描述要更加准确一些。

小可：可是我们只知道每一轮的情况，要求解复杂度的问题，还需要知道一共执行了多少轮次。

Mr. 王：很好，我们先来思考这样一个问题：每一轮的排序，使得已经有序的数组变得多大？第一轮是一个特殊的例子，我们让每一路内部实现了有序。每一个内部有序的组均有 M/B 个磁盘块。那么接下来的每一次，内存中都有 M/B-1 个磁盘块作为每一路的头。

小可：多出的那一个是缓冲区？

Mr. 王：对。此外，每一个头都带有一个含有 M/B 个磁盘块的有序的路，这样就实现了对 $(\frac{M}{B}-1)\cdot\frac{M}{B}$ 个元素的排序。第三次，我们用同样的方法，完成了 $(\frac{M}{B}-1)^2\cdot\frac{M}{B}$ 个元素的排序。

小可：那么究竟归并到什么时候算法可以结束呢？

Mr. 王：假设经过 k 轮之后，算法停止，可以列一个方程：

$$(\frac{M}{B}-1)^{k-1}\cdot\frac{M}{B}=\frac{N}{B}$$

小可：除了第一轮，我们都完成了 $(\frac{M}{B}-1)$ 路的归并。只要整个归并的数据规模达到了 N/B，就成功地归并了所有的元素。

Mr. 王拿出一张纸，说：来吧，发挥你数学基础的作用，求解 K。

小可在草纸上演算着，说道：两边取对数，以 $\frac{M}{B}-1$ 为底，结果就是 $k=\log_{\frac{M}{B}-1}\frac{N}{M}$。

Mr. 王竖起了大拇指，说：解得不错，我们简化一下这个结果，就是$k=\theta(\log_{\frac{M}{B}}\frac{N}{M})$。综合每一次的 I/O 复杂性和总次数，就可以求得整个算法的复杂度为$\theta(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{M})$。这个结果看起来依然比较复杂，该事实上，该式子可以简化为$\theta(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B})$。因为$\frac{N}{M}=\frac{N}{B}\cdot\frac{B}{M}$，加之此项在 log 内，是一个低阶项，在计算复杂度时可以直接忽略它。

Mr. 王：其实和内存算法中的快速排序类似，还有一个外排序版本的快速排序。就基本思想而言，和内排序一样。首先选出一个分界点，通过算法操作使得数组中左边的数都比它小，右边的数都比它大，然后对左边、右边分别执行这个步骤，不断地递归执行下去，就可以实现整个数组的排序了。

小可：哦，那么它的外排序版本又是怎么做的呢？

Mr. 王：快速排序的最重要动作就是划分。只是由于外存的原因，划分要更加有技巧，划分的终点，就是当每一块都可以被放进内存时。

还是用上面的例子来解释这个算法吧。

24 1 23 19 20 5 18 16 4 7 8 9

10 15 17 14 3 2 6 11 12 13 22 21

首先，在输入缓冲区中读入第一个磁盘块 24 1。

然后，为后面的 3 个磁盘块设置两个分界点。为了划分均匀，我们选用 8 和 16，至于如何选出 8 和 16 的后面再讲解。

现在内存中是 24 1,[][],[][],[][]。后面的 3 个磁盘块的两个分界点为 8 和 16。此时输入缓冲区里的数值是 24 和 1，我们分别将它们放进大于 16 和小于 8 的两个部分中，内存中变为：

[][],1[],[][],24[]

缓冲区已经空了，我们再将 23 和 19 放进来，23 大于 16，内存中换为：

[]19,1[],[][],24 23

此时我们发现 19 也大于 16，但大于 16 的块已经写满了，为 24 23，所以将 24 和 23 写回磁盘中，将它们在内存中占据的位置空出来：

[]19,1[],[][],[][]

再将 19 放进来：

[][],1[],[][],19[]

之后将 20 和 5 放进内存：

20 5,1[],[][],19[]

接下来是：

[][],1 5,[][],19 20

将 1 5,19 20 分别写到外存中。

……

不断执行上面的步骤，最终：

第一组　1 5 4 7 8 3 2 6

第二组　16 9 10 15 14 11 12 13

第三组　24 23 19 29 18 17 22 21

虽然组分好了，而且组间是有序的，但是每一组内部依然是乱序的状态，所以要继续对它们进行排序。

小可看了看写在纸上的数据，说：现在每一组都已经能够放入内存中了。所以对每一块用内存排序，就可以实现对整个数组的排序了。

Mr. 王：问个问题：假如执行过第一轮快排以后，划分的组仍然不能装入内存中，怎么办呢？

小可：只要对每一部分再执行一轮外排序就好了。如果还是不行，就再执行一轮。

Mr. 王：很好，这就是递归的体现。现在又到了分析复杂度的时候了，还是以 M、N、B 为参数。先来考虑：每一轮快速排序，I/O 复杂度是多少？

小可：这个好像比较复杂啊。

Mr. 王：想想看，虽然每个数据在内存中被拿来拿去，但是不论如何，它只是从缓冲区进来，直到该磁盘块被填满才离开内存。而离开的不会在本轮中再进来。

小可：我明白了，在每一轮中，每个数据都会进入到内存中一次，同时也都会离开内存一次，这说明 I/O 次数应该是一个 $\frac{N}{B}$ 的同阶函数 $\theta(\frac{N}{B})$。

Mr. 王：嗯，那我们一共要进行多少轮呢？第一次划分，我们将数据分成了 $\frac{M}{B}$ -1 组，这是因为内存中有 $\frac{M}{B}$ 个磁盘块，留下一个作为缓冲区，那么每一组中有 $\frac{N}{\frac{M}{B}-1}$ 个磁盘块。在第二次划分中，我们将每一组的 $\frac{N}{\frac{M}{B}-1}$ 个磁盘块再划分为 $\frac{M}{B}$ -1 组，每一组就有 $\frac{N}{(\frac{M}{B}-1)^2}$ 个元素。还记得要划分到什么时候为止吗？接下来的推导由你来完成吧。

小可：划分要持续到每一组都可以装入内存中为止，这意味着经过 k 轮的划分，应该达到 $\frac{N}{(\frac{M}{B}-1)^k}=M$ 的效果。把 k 解出来，就是 $k=\log_{\frac{M}{B}-1}\frac{N}{M}$。根据前面的内容进行处理，可以求得整个算法的复杂度为 $\theta(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{M})$。事实上，该式子可以简化为 $\theta(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B})$。这种形式和归并排序是一样的啊。

Mr. 王笑着说：哈哈，学得真快，非常好。

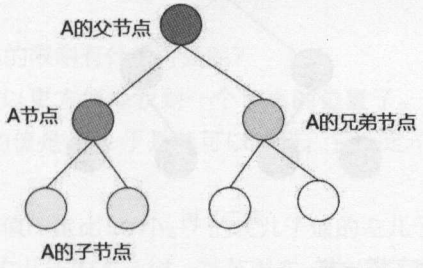
5.3　外存数据结构——磁盘查找树

5.3.1　二叉搜索树回顾

Mr. 王：接下来我们谈一谈外存查找结构。内存中的查找结构最典型的就是二查搜索树了。

这里我们先来简单地认识一下关于二叉树的问题。为了更好地理解在外存状态下的二叉树，必须要对内存中的树结构非常清楚。

一般意义上的树是一个图，像二叉树这种在计算机中用来存储数据的树型结构和一般的树是不完全一样的，它有一个根节点，而且它有一种自顶向下的方向性。我们一般也称这样的结构为“有根树”。对于一个一般的图来说，我们将与节点 A 有直接相连的一条边的节点都称作节点 A 的邻居。但在树型存储结构中则不然，与 A 直接相连、比 A 更接近根节点的节点，也就是 A 上一层中的节点称作父节点。与 A 直接相连的下一层中的节点称作 A 的子节点，A 的父节点的所有儿子，都是 A 的兄弟节点。而一个父节点的左儿子及其所有的后代，也就是父节点左边的这一支，我们称之为左子树。相应的，右边的那一部分，我们称之为右子树。



二叉树的结构

小可：哈哈，父节点和子节点这种叫法还真形象。

Mr. 王：在任何查找树中，每个节点都只有一个父节点。

小可：如果有多个父节点，就会产生环路，是吗？

Mr. 王：没错，反应很快，的确是这样，因为父节点的父节点不断回溯上去，都会汇集到根节点。也就是说，这两个父节点向上的方向是封闭的，如果这两个父节点还有一个共同的儿子，这就会出现一个闭合的回路。这就不符合树的定义了。

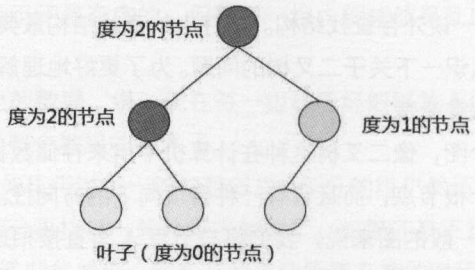
小可：之所以树被称为二叉树，是因为每个节点都分两个叉吗？

Mr. 王：顾名思义，但准确地说，二叉树中的每个节点最多有两个子节点，并不是所有的子节点都有两个儿子。不论如何，一棵树都会有叶子。

小可：这个“树”真形象，二叉树还有叶子啊？

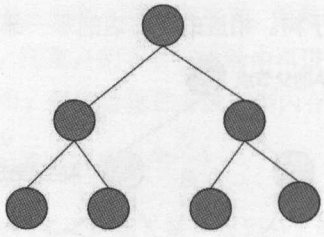
Mr. 王：树的叶子节点就是那些度为 0 的节点。注意，有根树的度和一般图的度是不一样的。在一般的图中，某个节点 n 的度是它邻居的个数；而在一棵有根树中，某个节点 n 的度是它的出度，也就是它直接后代的个数，或者说就是儿子节点的个数。

小可：嗯，那我就懂了，叶子就是没有后代的那些节点，只有一条边连向它的父节点。哈哈，这还真像一片叶子。



有根树中节点的度

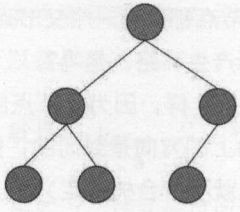
Mr. 王：如果一棵二叉树除了叶子以外所有节点的度都是 2 的话，它就是一棵满二叉树。



满二叉树

小可：嗯，长得像一个金字塔一样。

Mr. 王：如果一棵二叉树不是满二叉树，但是除了最底层以外，其他所有的层都是填满的，而且最底层节点连续集中排布在左侧的话，这样的二叉树就叫作完全二叉树。



完全二叉树

另外，还有一个概念就是树的高度。其实对于一棵有根树来说，从直观上讲，树的高度就是指这棵树有几层。节点在树中的高度，就是指从该节点向下直到某个叶节点的最长简单路径中边的条数。而一棵树的高度，就是指其根节点的高度。在一棵 k 叉有 n 个节点的有根树中，

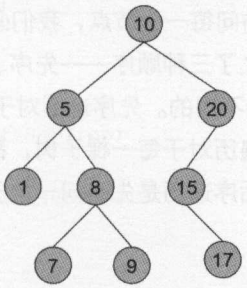
我们可以非常容易地发现，树的高度是 $\log_k n$。

小可：嗯，二叉树的高度就是 $\log_2 n$。

Mr. 王：根据我们前面学过的复杂度分析，可以发现在一棵 k 叉树上访问到一个确定节点的复杂度是 $O(\log n)$。这也是很显然的，因为途中要经过 $\log_k n$ 条边。

小可突然想到一个问题，说道：我记得前面还有一个概念叫作二叉搜索树，它和二叉树有什么不一样呢？

Mr. 王：二叉搜索树又叫二叉查找树。首先，它是一棵二叉树；其次，它满足这样一个限制，就是对于任意一个节点 n 来说，n 的左儿子值比 n 的值小，n 的右儿子值比 n 的值大。



二叉搜索树

小可：哦，可是做这样的限制有什么好处呢？

Mr. 王：这样我们就可以更方便地找到一个节点的位置了。比如要查找一个节点，其 ID 为 15。我们发现根节点 r 的值是 10，于是就可以知道，它一定不是其左儿子，或者其左儿子的所有后代。

小可：因为 r 的左儿子值一定比 10 小。r 的左儿子值的左儿子值一定比 10 还小。

Mr. 王：不难推出 r 的左儿子的右子树，其范围在 r 的左儿子值到 10 之间。这个你可以自己课后思考一下。

小可：的确是这样，所以它一定在它的右子树中。

Mr. 王：于是我们去访问它的右子树，右儿子值是 20。

小可：所以说 ID 为 15 的这个节点一定在右儿子的左子树上！

Mr. 王：没错，就按照这个方法继续访问下去，直到找到 15，或者停止在一个非 15 的叶子节点上，说明这棵二叉搜索树中不存在值为 15 的节点，返回不存在或者空即可。

小可：哦，这样加以限制之后，确实可以让计算机通过很简单的自动化步骤来找到一个特定值的节点。

Mr. 王：其实，这就是在二叉查找树上查找一个节点的算法描述。想一想，这个算法的复杂度如何？

小可：每一次查找，它都会沿着树向下深入一层。这就是说，最多不会超过树的高度，应

5.3.2 外存数据结构——B 树

小可：看来在磁盘上二叉搜索树对新来的数据插入树中支持得不好，那么究竟怎么解决这个问题呢？

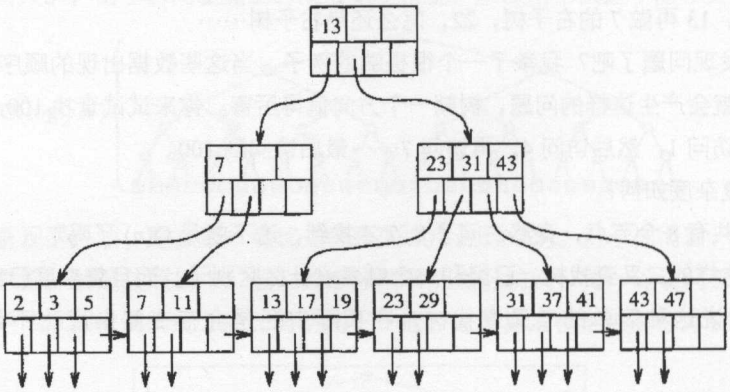
Mr. 王：有一个非常经典的解决方案，叫作 B 树。

小可：B 树？这个 B 树和二叉搜索树有什么区别呢？

Mr. 王：顾名思义，二叉搜索树是二叉的，这个 B 树就是 B 叉的。BFS 块自然对应于每个节点出度为 $\Theta(B)$ 的一棵树。而且在每一个磁盘块中，不放置来自树的多层的数据项，只放一层中的数据项。这样做的一大好处是，更新都可以通过变化节点的度来实现，此时我们进行树的平衡操作时，不再依赖旋转，而是利用节点的分裂和合并。

小可：太抽象了，这样说还是不太懂。

Mr. 王：好，我们来看一个例子。



这就是一棵 B 树。可以注意到，B 树上的每一个节点都有 k 个值和 k+1 个指针。每一个值两侧的指针，分别表示小于它和大于它的子树。比如 13 左侧的指针指向小于 13 的子树，13 和 17 之间的指针指向键值在 13 和 17 之间的子树，依此类推。

举一个查找的例子吧。当我们要查找键值 17 的时候，首先会和树根进行比较，17>13，所以要继续访问右子树，将其右子树的这个磁盘块读入内存进行处理，在右边的子树中发现 17<23，所以在 23 的左子树中，再将这一块读入内存中，然后在它的左子树的根节点中找到了键值 17。

小可：这样看来，B 树的查找效率很高啊。

Mr. 王：不过这里我们要研究一下，满足什么样性质的 B 树才能有这样高的查找效率。这里提出一个概念叫作 (a,b) 树。

小可：这个 (a,b) 分别表示什么呢？

Mr. 王：a,b 分别表示键值数量的上界和下界。

如果 T 是一棵 (a,b) 树（$a \geqslant 2$ 且 $b \geqslant 2a\text{-}1$），那么它应该满足下面三个条件：

- 所有的叶子在同一层上并且包括 a 到 b 个元素。
- 除了根节点，所有的节点的度在 a 到 b 之间。
- 根节点的度在 2 到 b 之间。

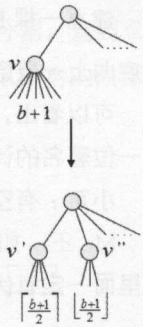
这里先以 (2,4) 树为例。

这样的树，在空间上是线性的，即 $O(N/B)$。

小可：这是为什么呢？

Mr. 王：首先，我们要存储的数据项有 N 个，将它们都存到叶子节点中。而每个节点的度均在 (a,b) 之间，即使在最坏的情况下，每个节点只有两个出度，此时内部节点的数量也是小于叶子节点的。更何况多数节点是在 a,b 之间的，内部节点数小于叶子节点数，说明它是线性空间的。

当所有节点的出度都是 a 时，树的高度 h 刚好是 $O(\log_a N)$，而节点的度普遍大于 a，所以 $\log_a N$ 是树高度的上界。同时我们选取 a,b 与 B 同阶，即 $a,b=\Theta(B)$，这样每个节点或者说每个叶子节点都能恰好装在一个磁盘块中。进行查找的时间复杂度就是 $O(\log_B N)$。



小可：这是因为 B 与 a 同阶吧？

Mr. 王：是的，我们进行了替换，使其与我们使用的参数相关联。

小可：到现在为止，我们研究的都是关于 B 树的性质和对建好的 B 树的使用，那么一棵 B 树是怎么建立起来的呢？

Mr. 王：好，接下来我们就谈谈 B 树的插入和删除。首先看插入，这是建立 B 树和新增数据项时维护 B 树的过程。

当某个节点容纳的数据项没有超过其限制 b 时，这种插入就很朴素了，我们讨论的关键是当节点 v 上面有 $b+1$ 个元素时，对 v 如何处理。此时我们要拆分节点 v，即创建节点 v' 和 v''，让它们分别有 $\left\lceil \frac{b+1}{2} \right\rceil \leqslant b$ 和 $\left\lfloor \frac{b+1}{2} \right\rfloor \geqslant a$ 个元素。

根据前面的约定，可以保证 $\left\lceil \frac{b+1}{2} \right\rceil \leqslant b$。但此时，消除了 v 却产生了两个新的节点，所以我们要把元素或指针插入 parent(v)。

这里还涉及一个问题，如果在拆分的过程中其父亲节点也填满了，那么还要再去拆分它的父亲节点；如果父亲节点的父亲节点也填满了，那么还要去拆分它的父亲节点的父亲节点。

小可：可是这一路拆分上去，恰好根节点也填满了，拆到了根节点怎么办？

Mr. 王：那就把根节点也拆了，使 B 树升高一层就可以了啊。现在想想，一次拆分会涉及多少个节点？

小可：自底向上的拆分和自顶向下的搜索应该是一致的，也是从树叶到达树根，最多再在树根上面加一层，所以都是 $O(\log_a N)$。

Mr. 王：很好，接下来我们讨论 (a,b) 树的删除操作。在正常情况下，我们将需要删除的元素从树中剔除就可以了。那么什么情况下会出现问题呢？

小可：我觉得和插入是同理的，当某个节点出现 a-1 个儿子节点时，就不行了。

Mr. 王：所以当 v 有 a-1 个元素 / 儿子节点时，将 v 和兄弟节点 v' 合并，将 v' 的儿子节点移到 v'，再从 v 的父亲节点中删除相应的元素和指针。一旦有需要，根节点被删除了也是有可能的，此时 B 树的高度会降低一层。

小可：我考虑到这样一种情况，如果合并之后又发现节点中的数据项数目大于 b 了，是不是还要拆分啊？

Mr. 王：没错，如果真的出现这种情况，还要按照插入的形式拆分 v，最终使得整棵树满足 (a,b) 树的性质。同样，这个过程也是要递归进行的，(a,b) 树也是涉及 $O(\log_a N)$ 个节点。

接下来我们来总结一下 B 树的性质。

B 树：(a,b) 树，其中 $a,b=\Theta(B)$。它满足 $O(N/B)$ 空间、$O(\log_B N)$ 查询、$O(\log_B N)$ 更新。另外，元素都在叶子节点中的 B 树有时被称为 B+ 树。

建立一棵 B 树需要 $\theta(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B})$ 次 I/O。这个复杂度包括排列元素、创建叶子节点，以及从底向上一层层建树。

可以看出，B 树是一种性质非常好的数据结构，其广为做磁盘算法的计算机科学家所热爱。有一位著名的计算机科学家，更是在他的主页中写道：我喜欢 B 树。

小可：有空我是不是也应该试着去实现一棵 B 树呢？

Mr. 王：如果是要在开发软件系统中使用 B 树，不妨参考一些开源关系数据库的源代码，那里面一定有优质的 B 树的实现。

5.3.3 高维外存查找结构——KD 树

Mr. 王：以往我们在数据结构中进行的查找，都是查找某一个键值或者某一个区间内的值，这样的查找称之为一维查找。

小可：难道说还有多维查找吗？

Mr. 王：现在我们就来介绍一种高维查找结构——KD 树。

小可：可是什么样的查找是高维查找呢？

Mr. 王：举个简单的例子。你平时会用到位置服务的 App 吗？

小可笑着说：我今天中午还用大众点评查找过周围的饭店，饱餐了一顿呢。

Mr. 王：你的位置在定位系统和定位服务中就是一个坐标，这个坐标就是一个二维数据项。你在查找周围的饭店时，就已经进行了一次二维空间内查找。我们现在要考虑的，就是如何用

计算机存储这些二维点，并且可以以非常高的效率查找出来。

小可：原来是这样。那么如何来实现二维空间内的高效查找呢？

Mr. 王：计算机工作者们曾经提出过很多种二维空间内查找的方法，像网格文件、R 树、四叉树等，在实际应用中使用最多的应该是 R 树。今天我们要讲的二维空间内查找结构叫作 KD 树，之所以讲它，是因为它的性质比较容易保证。

小可：那什么又是 KD 树呢？

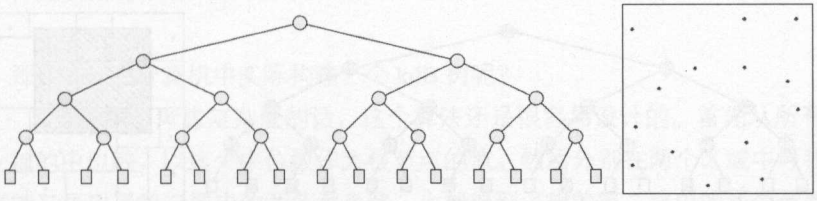
Mr. 王：简单来说，KD 树很像是将两个二叉树叠在一起。考虑一下：当我们进行一维数据查找时，需要在一维上对数据进行索引，建立的就是一棵二叉查找树；而当我们要查找的点是一个二维数据时，就需要在两个数据维度上都建立索引，这时就需要两棵二叉树，这两棵二叉树分别索引的是 x 轴和 y 轴，我们可以在两棵轴上面进行二分搜索。而将两棵二叉树的层次交替存储，就合并成了 KD 树。

小可：KD 树具体是如何定义的呢？

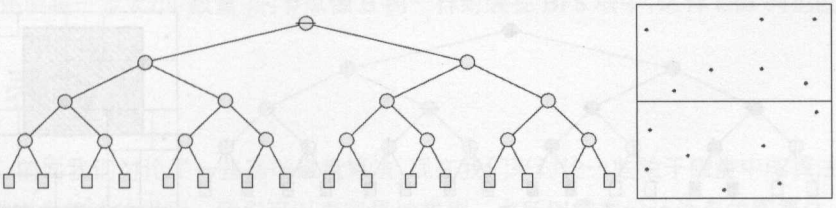
在一棵 KD 树上，我们用树的偶数层中的节点来表示空间中的水平线；相应地，我们用奇数层中的节点来表示空间中的垂直线；这些垂直线和水平线会对整个区域进行分割，直到点集被划分为每个区域内只有一个点为止。那么水平线和垂直线也就相应地对应着 KD 树的内部节点，而在二维平面上，我们要检索的这些点就对应着 KD 树的叶子节点。

小可带着疑惑的表情说：我还是不太明白。

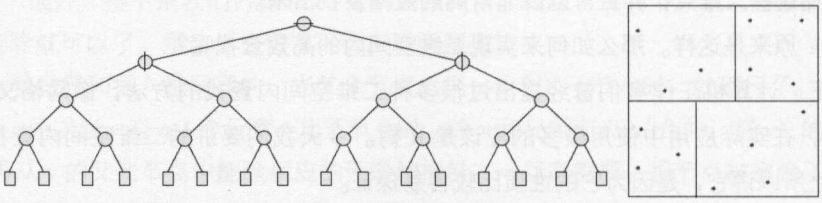
Mr. 王：我们来举个例子吧。



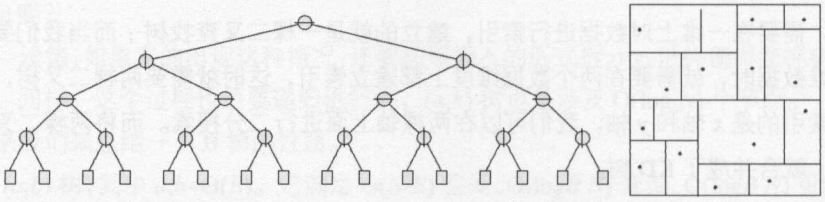
左边是一棵 KD 树，右边是一个二维平面。下面我们分步演示算法的过程。



我们将树根定义为一条水平线，在区域中画下它代表的水平线。



下一层中的节点代表的是垂直线，我们在图中标示出这两条垂直线。

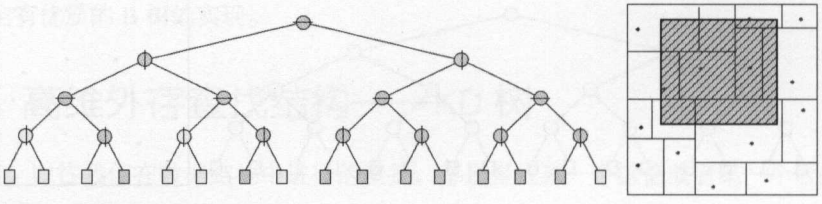


依此类推，这样所有的点都被放进了单独的一个区域里。也就是说，每一片叶子都已经仅仅代表一个节点，KD 树就建好了。

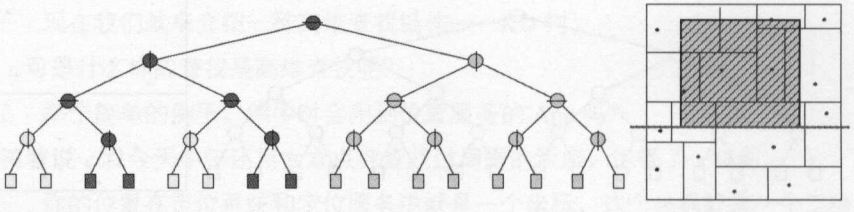
小可：哦，这样我就明白多了。那么对一棵建好的 KD 树又是怎样进行查询的呢？

Mr. 王：在查询一棵 KD 树时，我们会递归访问节点的相应交叉查询区域，并且报告在树 / 节点中且在查询中完全包含的点。

这样说太抽象了，我们还是举个具体的例子吧。看图中的绿色区域，在这个检索中，我们希望找出灰色区域中的点。



首先我们来看灰色区域的下界。



对一棵 KD 树来说，它的根是一条水平线，我们就可以根据灰色区域的下界画一条水平线。

然后比较这条水平线和根的高低，在 KD 树上，就是比较树根代表的水平线的高度值和检索区域的高度值。比如在这个例子中，我们发现根节点的高度比下界要高，这说明根节点的右子树一定都是候选集合，而根节点的左子树只有一部分是候选集合。但是为了确定这一部分，我们就要访问它的左子树。然而树的第 1 层（根是第 0 层）是用来表示垂直线的，我们无法用它来判断水平维度的高低。我们再去访问第 2 层，在第 2 层中，可以用这个下界和第 2 层中的两个节点进行比较，从而得出下一层将要访问的节点，我们继续向右子树查找。

同理，我们可以不断地用区域的四个边界在树上进行查找，对于树中的不同层次交替根据纵横划分进行查找，直到最终确定绿色区域内部所有的点，也就是 KD 树上的叶子节点。

现在我们来考虑一下 KD 树的查询效率如何。

现在你觉得这棵树是不是适合磁盘存储呢？

小可：虽然 KD 树来用特殊的设计有效地表示了空间中的二维点，在设计思想上非常的巧妙，但是从本质上说，依然是一棵二叉树，依然存在着二叉树不适合存储在磁盘上的问题，比如有旋转调整这样的麻烦。

Mr. 王：的确。由于 KD 树同样具有二叉查找树的种种性质，所以也就同样存在二叉查找树在磁盘上存储种种不方便的问题。不过正是由于它和二叉树相似，所以我们不妨也采取和二叉树相似的改进方法。为了将查找树结构引入到磁盘上，我们引入了 B 树。这次我们也可以发展 KD 树，引入一种适合存储在硬盘上的数据结构——kdB 树。

小可：kdB 树是不是就是把 KD 树和 B 树融合到一起啊？

Mr. 王：是的，kdB 树结合了 KD 树和 B 树的思想，使得 KD 树更加适合磁盘存储。在具体的实现中，逻辑结构依然采用 KD 树，当叶子包含 $B/2$ 到 B 个点时停止分割，在内部节点形成 BFS 块。

小可：那么如何在计算机中实际构建一个 kdB 树呢？

Mr. 王：其实如果不考虑复杂度的话，这个算法还是很容易设计的。首先从所有的点中找到纵坐标 y 轴的中位数，以这个中位数作为根节点的值。然后分别在两个区域中寻找 x 轴的中位数，这样就又画出了第二层中的两条垂直线，也就得到了树的第二层中的两个节点的值。依此类推，递归地在新划分出来的区域中交替寻找 x 轴和 y 轴的中位数，这样 KD 树就建好了。当然，我们还要将一定大小（数量）的节点像 B 树一样封装在 BFS 块中，这样 kdB 树也就建好了。

5.4　表排序

Mr. 王：前面我们讨论了一些基础磁盘算法，现在我们来讨论一些关于磁盘中图算法的问题。通过对基础磁盘算法的学习，我们可以很容易地想到，之所以需要设计外存的图算法，是因为如果内存无法存储全部的数据的话，我们就要尝试将数据存放在外存中；图也是一样的，当需

要表示的图很大时，内存无法存下全部的图节点或者边时，我们就要尝试将数据保存在外存中，仅当需要对图的某一部分进行处理时，才加载到内存中来。

图算法的体系是比较庞大的，对图的操作和研究的算法也是非常多的，在开始研究一些比较复杂的图算法之间，我们先来讨论一个基础的算法，叫作“表排序”。

小可：排序？是对一张表里面的数据进行排序吗？用前面的归并排序法可以解决吗？

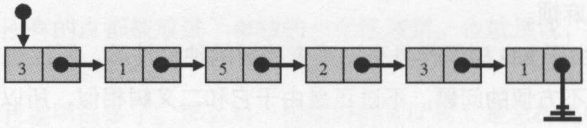
Mr. 王：这里的排序和前面的不太一样，我们称前面的排序为“sort”，称现在要讲的这种排序为“ranking”。

小可：这个 ranking 具体是做什么的呢？

Mr. 王：链表的 ranking，求解的是在一个链表中，某个节点到表头节点的距离。

小可：如果是这样的话，那么第一个节点就是 1，第二个节点就是 2，第 n 个节点就是 n 呗？

Mr. 王：如果节点没有权值的话，的确是这样的。但是当我们要应用它时，往往情况更加复杂，链表中的每个节点往往都带有一个权值，计算时要带上这个权值。



就比如上面这个图，它的每个节点都带有一个权值，那么第一个节点的 ranking 就是 3，第二个节点的 ranking 就是 3+1=4，第三个节点的 ranking 就是 3+1+5=9……

小可：原来是这样的排序啊，但是也太容易了。我只要从头至尾进行一次扫描，添加一个前面的距离已经达到多少的计数器不就可以了吗？

Mr. 王：没错，如果该链表可以全部放入内存中的话，这个问题确实非常简单，进行一次扫描就可以完成。但是如果这个链表中的节点很多，并且节点很大，每个磁盘块只能容纳几个节点的话，我们就要把这些节点存放在外存中；而且如果节点是随机存储的，也就是它们在磁盘中存放的物理位置与其在链表中的逻辑位置没有明显的关系，情况就会比较复杂。

比如每个磁盘块可以存下 4 个节点，内存中可以放置 2 个磁盘块，磁盘块上面的数字表示其在链表中的顺序，就像下面的图一样。我们模拟运行一下看看会出现什么样的问题。

第 1 步：我们访问节点 1，将它所在的 1 号磁盘块加载进内存中。

第 2 步：我们访问节点 2，将 2 号磁盘块放进内存中。

第 3 步：我们访问节点 3，这时内存中存放着 1 号和 2 号磁盘块，已经被填满，必须换出

一些磁盘块，腾出一些内存空间以加载新的磁盘块。假设我们采用先进先出的策略，将 1 号磁盘块从内存中删除，然后加载 3 号磁盘块。结果就是：

第 4 步：我们访问节点 4，内存依然是满的，将 2 号磁盘块从内存中删除，然后加载 4 号磁盘块。

通过算法的第 3 步和第 4 步就可以看出，从内存被填满的下一步开始，每一次访问都要调入一个磁盘块并丢弃一个磁盘块。这也意味着每处理一个节点都要进行一次磁盘 I/O，如果按照这样的方法访问下去的话，在整个算法的运行过程中，磁盘 I/O 的次数已经接近了节点的总量 N。也就是说，算法的最坏情况的 I/O 数为 $\Omega(N)$。

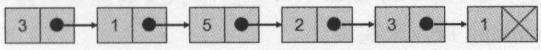
小可：嗯，这么大的 I/O 数肯定会导致程序运行的速度特别慢，用户肯定无法接受。

Mr. 王：现在看来，表排序这个问题并没有那么简单了吧。所以我们需要想一个面向外存的办法来解决这个问题。这里给出一个高效的表排序算法。

这个算法是这样做的：首先，我们取一个独立集，这个独立集的大小至少是 $N/3$。

小可：什么是独立集呢？

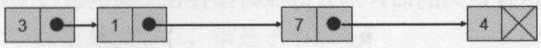
Mr. 王：一个图中的独立集，是一些两两之间不存在边的点的集合。独立集怎么获得，我们之后再去讨论。现在假设我们有一种方法来获得这样一个独立集。



比如我们取这样一个链表，找出一个包含 5 和 3 这两个节点的独立集。然后，去掉独立集中的点，对剩下的这个子链表的 ID 进行排序。将排序结果放在连续的磁盘块中，而将独立集中的元素按照其后继节点（也就是它们后面的那个节点）的 ID 进行排序，放在一组连续的磁盘块中，这样这两部分都按照 ID 有序地放置在连续的磁盘块中了。想想看，这样做有利于接下来进行一个什么样的操作？

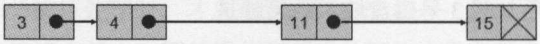
小可：两组有序的表，还是整个链表的片段，那么就可以进行归并？

Mr. 王：非常好，的确，这样做的目的就是非常有利于实现归并操作。另外，对独立子集中的部分我们按照其后继节点的 ID 进行排序的好处是，方便将它的权值加到其后继节点中。



此时，我们已经将独立集中的节点所带有的 5 和 3 这两个权值加入到了它们的后继节点 2 和 1 中，这两个节点的权值变成了 7 和 4。这样我们就得到了一个独立集中的节点被去掉，但却不影响最终 ranking 数值结果的链表片段。

现在我们就可以在这样的链表片段中做 ranking 操作。求出的结果是：



最后我们再做一个工作，就是把独立集加回来，加回来这个工作就是按照 ID 进行归并排序。原来的链表片段中的 ranking 已经不用动了，因为我们考虑了独立集中的值，所以它们的值的结果是正确的，此时只需要相应地更新一下独立集里的 ranking 就可以了。这个操作非常简单，只需要将其前驱节点的 ranking 加上节点值即可。这样做的好处主要是达到了压缩 L 的目的，极大地降低了 I/O 复杂度。

5.5 表排序的应用

5.5.1 欧拉回路技术

小可：我还有一个问题：今天我们不是要讨论关于磁盘的图算法吗？可是花了好大的劲一直在讨论链表啊？

Mr. 王：其实可以想想，链表本质上也是图，只不过这种图非常特殊，是线性的。不仅如此，这种表的 ranking 其实是有很大的实际应用意义的，很多复杂的图算法都是以链表 ranking 作为基础或者子操作的。比如，如果我们能够尝试将一棵树 T 用链表 L 来表示的话，那么对 T 的很多操作都可以用对 L 的 ranking 操作来实现。

小可惊讶地说：真的吗？那树如何用表来表示呢？

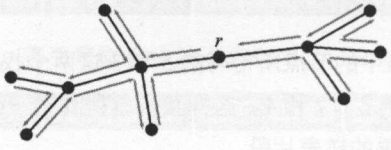
Mr. 王：别急，首先我们要了解一种方法，叫作欧拉回路技术。

小可：是一笔画吗？在图中“一笔画”的轨迹就是一个欧拉回路。

Mr. 王：其实很多树由于没有圈的存在都是不能一笔画的，就比如一棵“Y 字形”的树，就不能一笔画，也就不存在欧拉回路了。但是我们在树的范畴中要重新定义一种欧拉回路，称之为“树的欧拉回路”。

树的欧拉回路定义为：从树中的某一点出发，经过任意一条边两次，最终回到出发点的这样一个回路。

小可：哦，如果是这样定义的话，那么每棵树都存在“树的欧拉回路”了。



Mr. 王：比如对于这样的一课树，浅色线条就是它的一个欧拉回路，从 r 点出发，经过它的每一条边两次，访问整个图，回到 r 点。

在内存中，实现树的欧拉回路求解是非常容易的，只要沿着点一个个地搜索下去，就可以很容易地得出结果。不过，如果树中的节点是无明确规律、随机地存放在磁盘中的，那么就不那么容易了。这时候，如果我们希望用比较高效的方法来解决这个问题的话，就要考虑能否只根据本地的信息。也就是说，对于每个节点，我们只研究它自己和与它相邻的那些边和节点，而不必去考虑树上与之距离较远的其他部分，来求出欧拉回路。事实上，这是可以实现的，而且实现效率也很高。

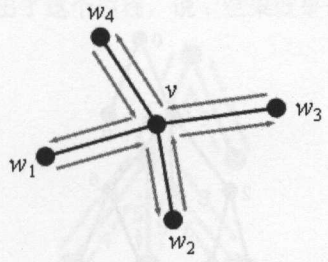
小可：如果是这样的话，计算的压力确实可以小很多，那这种思想具体是怎么实现的呢？

Mr. 王：首先，我们要对这棵无向的树做一个小小的变化，将每一条无向边变成两条有向而且方向相反的边的组合。比如 (v,w) 这样一条无向边，我们将其记为 (v,w) 和 (w,v) 两条单向边的一个组合。这是因为在树的欧拉回路中，每一条边都要走两次，并且一定是方向相反的。

然后，还要进行一个转化，就是将一条边看作是一个链表中的数据节点。也就是说，与这棵树 T 等效的链表 L 中的每一个数据节点，都是我们前面定义的一条有向边。

小可：那这些边在链表中的连接关系又是怎样的呢？

Mr. 王：嗯，我们来看这个图，假设 v 顶点有多条与之相连的边，分别为 $\{v,w_1\},\{v,w_2\},\cdots,\{v,w_r\}$，我们就将链表 L 中 $\{w_i,v\}$ 的后继表示为 $\{v,w_{i+1}\}$。也就是说，$\{w_1,v\}$ 的后继是 $\{v,w_2\}$，$\{w_2,v\}$ 的后继就是 $\{v,w_3\}$……这样就能保证对于任意一个节点 v，与之相连接的节点 w 都能有一去一回的路径。



小可：这样的算法的确能够实现求解一棵树的欧拉回路，同时还能将一棵树 T 完全表示成一个链表 L。但是这样的算法求解出来的链表 L 能够体现树 T 的性质吗？比如我想知道，在原来的那棵树 T 上，两个节点谁是父节点，谁是子节点呢？

5.5.2　父子关系判定

Mr. 王：这样的问题叫作父子关系判定。父子关系判定，就是求在有根树上，两个有一条

边相连的节点，谁是父节点，谁是子节点。这在内存中是非常容易判定的，只要从根节点出发进行一次先序遍历，就可以很轻松地解决。但是在外存中则不然，因为在磁盘中相邻的点是不一定被放在一起的，查找节点的难度会变大，两个在磁盘中不连续存放的节点就需要一次磁盘I/O，就需要考虑I/O复杂度的问题。

具体的解决办法就是，可以利用我们前面提到的欧拉回路技术。首先，我们以树的根节点为起点，根据树的欧拉回路创建与之等效的链表 L。然后，根据下面的条件进行判断：

$$v = p(w)，当且仅当 \text{rank}((v,w)) < \text{rank}((w,v))$$

小可：这就是说，如果(v,w)这条边的rank小于(w,v)这条边的rank，那么v一定是w的父节点。这是为什么呢？

Mr. 王：因为我们是从根节点出发的，如果v是w的父亲，那么一定会先访问到v，再访问到w，所以先走的就是(v,w)这条边，等到回来时再走(w,v)，故$\text{rank}((v,w)) < \text{rank}((w,v))$。

小可：嗯，如果已经知道了一棵树上节点间的父子关系，那么就更有利于掌握树的结构了。

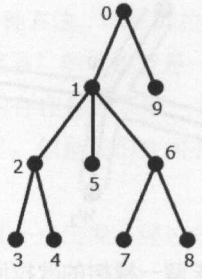
5.5.3 前序计数

Mr. 王：我们再来说说父子关系判定的应用。前序计数是一种非常常用的对树进行处理的方法。前序计数实现的就是对各个节点按照其前序遍历的序列进行标号，第一个访问的记为1，第二个访问的记为2，依此类推。

小可：嗯，这个操作在内存中同样也是非常容易实现的，只要前序遍历一次树就可以了。

Mr. 王：在磁盘中，我们依然可以借助欧拉回路技术和将树存储为链表这种策略。

比如对于这样一棵树：

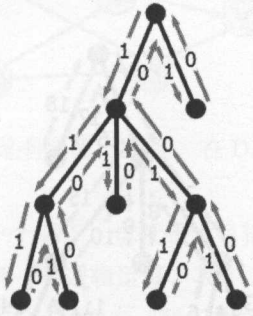


图中的数字就是其前序遍历的顺序。现在我们要对存在磁盘中的这样一棵树的节点求解出它的前序计数。想一想，如果不采用任何面向磁盘的特殊设计，而是采用朴素的搜索算法的话，复杂度会怎么样？

小可：我认为和前面的磁盘中的链表相类似。这些节点放置于随机的磁盘块中，当内存满了以后，在最坏情况下每次访问一个节点都要换入一个新的磁盘块，这样就造成了$\Omega(N)$的复

杂度。这样的复杂度是不能接受的。

Mr. 王：那我们就来设计一种比较好的方法。我们完全可以利用前面的欧拉回路技术，但是在求解这个问题时，要做一个特殊的处理。



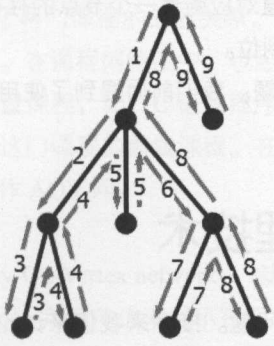
在每一条边上，我们将从父节点指向子节点的有向边的权值设为 1；反之，将从子节点指向父节点的有向边的权值设为 0。

小可：父节点和子节点的判定刚好可以利用前面的父子关系判定！

Mr. 王：没错，这样欧拉回路构成的链表在顺序访问时，就会在从父节点向子节点遍历时增加 1，这是在前序计数时我们所需要的；而在从子节点返回向父节点移动时，不增加值。在这个实例中，我们可以非常容易地得出结果，每一个节点的前序计数结果都是 (parent(v),v) 这条边的 rank，也就是：

$$\text{preorder}\#(v) = \text{rank}((p(v), v))$$

小可拿出纸笔，在纸上写画出了这个过程，说：结果就是这样：

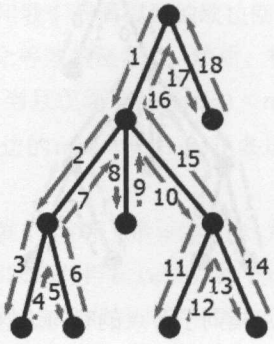


Mr. 王：这个过程的时间复杂度就是 O(Scan(N))+O(Sort(N))=O(Sort(N))。和进行表 ranking 的复杂度一致，相比 $\Omega(N)$ 而言，真是快得太多了。

Mr. 王：还有一类问题叫作求子树大小。求子树大小就是在树的每一个节点上标出其子树

上节点的个数。

这一次，我们依然采用欧拉回路技术。在从父节点去子节点的路上，我们依然在边上标注1；不同的是，在回来的路上，我们同样将权值设为1。这样，经过任何一条有向边，都会让ranking的计数加1。就像这样：



那么每一个节点的子树大小为：

$$|T(v)| = \frac{\text{rank}((v, p(v))) - \text{rank}((p(v), v)) + 1}{2}$$

你来思考一下，为什么是这个数?

小可想了想，说：$\text{rank}((v, p(v)))$这个数描述了当遍历到达它时，已经经历过的边数，$\text{rank}((p(v), v))$描述了当遍历从v的子树结束返回v时ranking的累积，两个rank作差，可以求出在这个子树中，究竟走了多少条边。加上1是为了调和v自己本身；而除以2，是因为每一个节点的存在，都有一去一回两次。也就是说，rank这个值会因为一个节点的存在而增加2。所以只要除以2，就可以让每个权值对应衡量一个节点的存在。

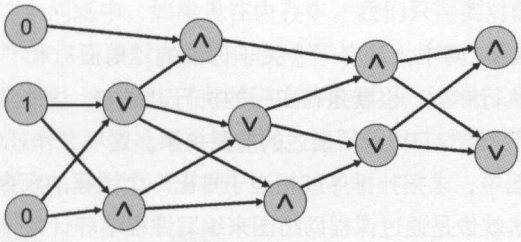
Mr. 王：非常好，你的解释很到位。

小可：对了，我想起来一个问题。我们前面提到了使用最大独立集，但还没有说怎么求最大独立集呢!

5.6 时间前向处理技术

Mr. 王：很好，你还记得这个问题。接下来我们来讨论另一种磁盘中的大数据算法策略，叫作时间前向处理方法。在这种策略中，我会讲解求解最大独立集的方法。

先介绍一个时间前向独立集的其他例子。



这是一个 DAG。所谓 DAG 就是有向无回路图。在 DAG 中的每一条边都是有方向的，但是 DAG 中不允许有环形的回路。

这个 DAG 比较特殊，它的每一个源点，也就是没有指向它的边的节点都有一个逻辑变量值 0 或者 1。所有的中间节点上都有一种逻辑运算，其中 ∧ 是与运算，∨ 是或运算。逻辑运算的计算方法你应该比较清楚了。在这个 DAG 中，我们希望求解所有的点上代表的逻辑变量值。比如某一个顶点 v 上面的运算是与运算，而指向它的两个逻辑变量值均为 1，那么顶点 v 所代表的逻辑变量值就是 1。依此类推。

解决这个问题的关键之一是，我们必须要知道按照什么顺序来计算这些点的逻辑变量值。如果当我们要计算一个点的值时，它的两个输入的值还没有求出来，那么这个点的值就无法求出来，需要等待前面的节点的值。所以，我们要先理解一个内存算法，叫作拓扑排序。

小可：拓扑排序?

Mr. 王：是的，拓扑排序是一个非常经典的图算法，适用于 DAG，将 DAG 转化为一个序列。

首先我们来谈谈什么是拓扑排序。你听说过课程网络图吧?

小可：就是描述那些课程前置关系的有向图。比如要想学高等数学，就应该先学初等数学。在课程网络图中，就画一条有向边，由初等数学指向高等数学。

Mr. 王：嗯，这条边也就描述了两门课程的先后关系。在实际生活中，类似课程网络图这种具有先后关系的例子是非常多的。在课程网络图中，也经常会出现一些比较复杂的情况，比如两门课程同时是第三门课程的前置课程，如“C 语言程序设计”和“计算机数学基础”这两门课程，都是“数据结构与算法”这门课程的前置课程。在实际的课程图中，这个网络会变得非常的庞大且复杂。这种网络也叫作 AOV 网。

小可：什么是 AOV 网?

Mr. 王：所谓 AOV 网（Activity-On-Vertex network），就是顶点活动网。在这种网络中，我们用节点来代表一个活动（Activity），用有向边来表示它们之间的前置关系。在课程网络图中，这个 Activity 代表的就是课程，它用图中的节点来表示，而用有向边来表示课程中的前置关系。比如有两个节点 A、B，如果有一条边 (A,B) 就说明 A 必须在 B 之前执行，或者说仅当 A 执行过之后，B 才能执行。AOV 网具有这样一个特点，就是它是一个 DAG，也就是有向无回路图。

小可：嗯，如果有回路就不对了，“后面”的课程不可能重新成为“前面”的课程的前置。

这是符合逻辑的。

Mr. 王：没错，在 AOV 网中，就有一个关于你说的“前面”和“后面”的问题。我们想要排出一个各项任务的执行序列，也就是在实际的执行过程中，这些活动到底谁先执行，谁后执行，同时要使得它们不违背 AOV 网所描述的前置关系。这项工作就叫作拓扑排序。

小可：在课程网络图中，求拓扑排序就相当于排出一个课程的顺序表吧，就是我先上哪一门课，再上哪一门课。这就像是通过课程网络图来编写课程培养计划一样。

Mr. 王：没错，现在我们来形式化这个问题。

拓扑排序就是将像 AOV 网这样的 DAG 转换成一个线性序列，使得如果 AOV 网中存在一条边 (a,b)，那么在形成的这个线性序列之中，a 就要出现在 b 的前面。

小可：这个问题还真的很有意义，可以用来排前面提到的课程顺序表。那么这个问题怎么解决呢？

Mr. 王：其实有一个非常简单易行的算法可以解决这个问题。

每一轮的迭代只需要完成三项工作：

第一，找到一个没有入度的顶点。

第二，将它插入到拓扑序列中。

第三，将它和它所有的出度从 AOV 网中删除。

小可：就这么简单？

Mr. 王：没错，就是这么简单。我们可以用刚才的图来举例。

其实它的原理也是比较容易想通的。首先，任何一个没有入度的节点，它就处在了“可执行的状态”。

小可：是，它没有任何前置活动的限制，可以执行。

Mr. 王：所以我们就可以把它插入到执行序列中。当一个节点被插入到执行序列中时，我们就可以认为它在接下来的活动执行之前已经被执行完了，而且如果它执行完了，它将不再影响其后置活动的执行。比如学过了“计算机数学基础”和“C 语言程序设计”，就可以学习“数据结构与算法”了，所以我们可以将加入了拓扑序列的那些节点的出度删除，然后这个节点也就没用了，同样删掉。

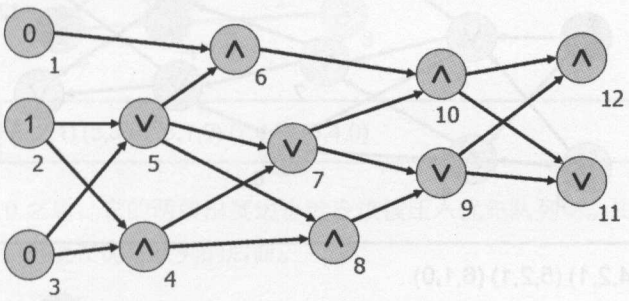
小可：于是就会有更多的节点“露”出来，没有了入度，它们就可以被加入到拓扑序列中。我懂了，这个步骤可以持续执行下去，直到所有的活动都被加入到拓扑序列中。可是在每一个步骤中，如果发现有两个节点都没有入度怎么办呢？

Mr. 王：这种情况也是普遍存在的，在这种情况下任选一个或者取 ID 较小的那个节点就可以了。因为选择任何一个节点都不会破坏拓扑序列满足 AOV 网的要求。但这也意味着，拓扑序列是不唯一的。

小可：嗯，我懂得拓扑排序了。

Mr. 王：我们回到这个问题中，如果是在内存中，我们只需要对前面的这些点做一个拓扑排序，就可以保证每一个节点在求解时，它们的所有入度节点都已经求出来了。

我们现在要考虑的就是，这种方法在磁盘中如何去实现。首先，我们要先将所有节点的拓扑序列求出来，如图中的标号，就是其拓扑序列。这里采用的就是前面说到的时间前向处理方法。我们借助一个数据结构，叫作优先队列。



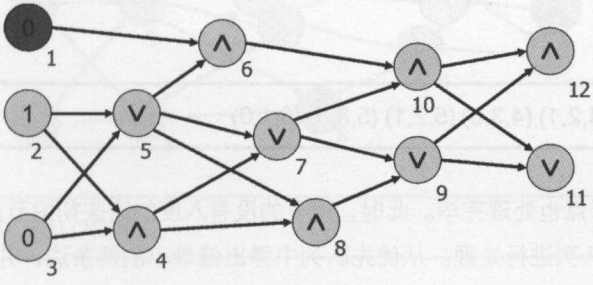
小可：队列是满足先进先出策略的一个线性结构，那优先队列是什么？

Mr. 王：优先队列满足这样一个条件：优先队列中每个节点都有一个值，以任意顺序入优先队列的节点，会以值从小到大的顺序出队列。优先队列的内部是一个堆，今天我们先不谈其内部实现，你只要知道优先队列的出队列顺序，与其值的大小有关，值小的先出队列，值大的后出队列，而不是入队列时的顺序就可以了。

Mr. 王：在这里我们使用的优先队列中的节点不是顶点，而是边。每条边的记法为（终节点的拓扑编号，始节点的拓扑编号，始节点上的布尔变量）这样的一个三元组，比如 1 号节点到 6 号节点这条边记作（6,1,0）。而在优先队列中，我们用来做比较的值是终节点的拓扑编号这个数据域。也就是说，被压入优先队列的节点的拓扑编号越小，在出队列时就会先取出它。

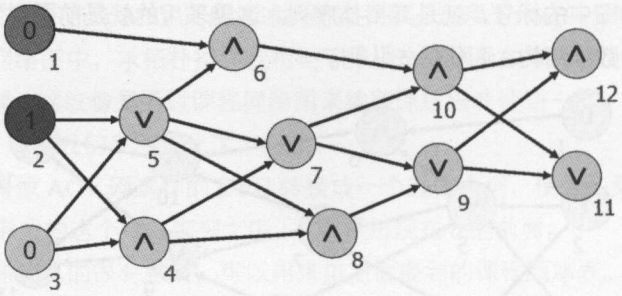
小可：那么算法具体是怎么运行的呢？

Mr. 王：还是用刚才的例子。



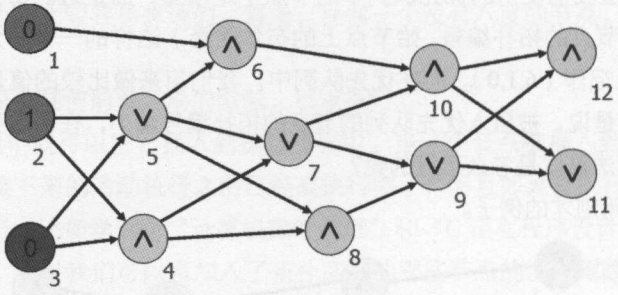
Q: (6,1,0)

第 1 步：我们将 (6,1,0) 这个数据记录压入优先队列中。有关 1 号节点所有的边，我们都已经处理完毕了。



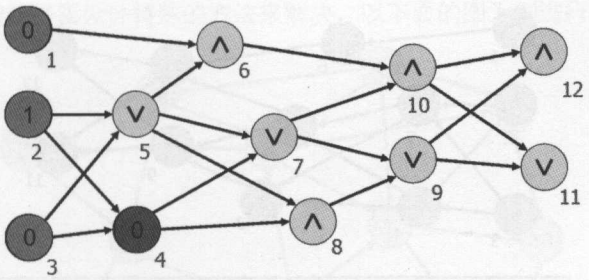
Q: (4,2,1) (5,2,1) (6,1,0)

第 2 步：我们处理 2 号节点，方法是同样的。注意，2 号节点有两条边，这两条边都要处理。另外，注意优先队列中的点，它们会按照第一个数据域“终节点的拓扑编号”进行排序，以便最后按照从小到大的顺序出队列。虽然 (6,1,0) 是先入队列的，但是 4 的值比它们小，所以 (4,2,1) 要排在前面。



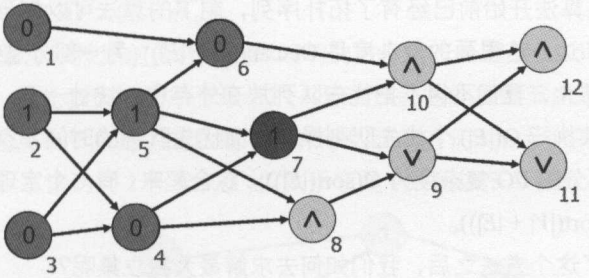
Q: (4,2,1) (4,3,0) (5,2,1) (5,3,0) (6,1,0)

相应地，3 号节点也处理完毕。此时，所有的没有入度的这些初始节点都已经被处理完毕。下面我们沿着优先队列进行处理。从优先队列中弹出值最小的两条边，并且进行处理，本例中是 4 号节点的两个入度边，源点分别为 2 和 3，值为 1 和 0，计算可得 4 的值是 0。

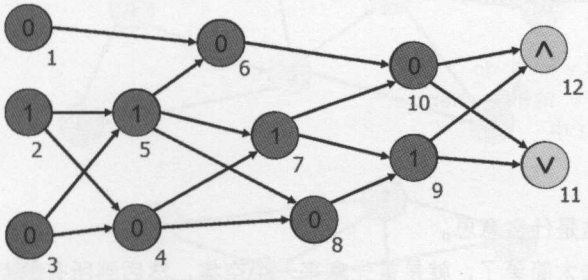


Q: (5,2,1) (5,3,0) (6,1,0) (7,4,0) (8,4,0)

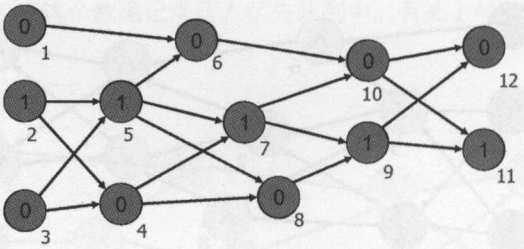
求出 4 的值是 0 之后，它的两条出度边也就应该被压入优先队列中。由于 7 和 8 这两个值是最大的，所以它们出现在优先队列的后面。



Q: (8,4,0) (8,5,1) (9,7,1) (10,6,0) (10,7,1)



Q: (12,9,1) (12,10,0)



Q:

接下来，该算法用同样的方法进行计算，直至优先队列被最终清空时，算法停止。此时，所有节点上的值都已经有了结果。

小可笑着说：又到了分析复杂度的时候了。

Mr. 王：假设在算法开始前已经有了拓扑序列，剩下的算法可以划分为两部分，其中一部分是扫描所有的点和边，这里面的复杂度是 O(scan(|*V*| + |*E*|))；另一部分是维护优先队列的开销。由于这个数据量非常大，我们不得不把优先队列放在外存中。统计一下，每条边都要从优先队列中进出一次，总共执行 O(|*E*|) 个优先队列操作，而优先队列的时间复杂度与对其排序的下界是一致的，所以这部分的 I/O 复杂度为 O(sort(|*E*|))。综合起来，有一个定理：一个 DAG $G = (V,E)$ 其 I/O 复杂度是 O(sort(|*V*| + |*E*|))。

小可：嗯，有了这个方法之后，我们如何去求解最大独立集呢？

Mr. 王：好，现在我们来谈谈最大独立集的问题。首先求解最大独立集是一个 NP-hard 问题，接下来要介绍的这个求解方法是一个近似算法，不是精确解，因为求解精确解的开销过大。

伪代码算法 GREEDYMIS：

```
1  I = 0
2  for 每个顶点 v ∈ G do
3   if I 中没有 v 的邻居 then
4     将 v 加入 I 中
5   end if
6  end for
```

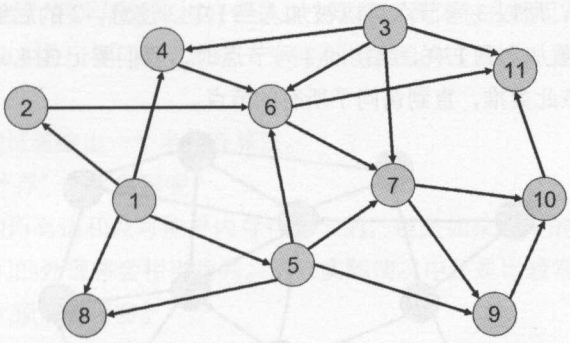
你来解释一下这是什么意思。

小可：这个算法太简单了，就是首先拿来一个空集，然后到所有的点的集合中去找，只要 I 中还没有 v 的邻居，就把 v 放进集合中。这个算法很朴素啊！

Mr. 王：没错，这是一个非常朴素的贪心算法。在内存中求解其最大独立集非常容易，但是在外存中会存在什么问题呢？

小可：显然，由于要扫描所有的节点，所以仍然非常容易出现这些节点因存储没有规律，导致其随机地存放在不同的磁盘块中，最后以 Ω(*N*) 的顺序结束扫描的情况。

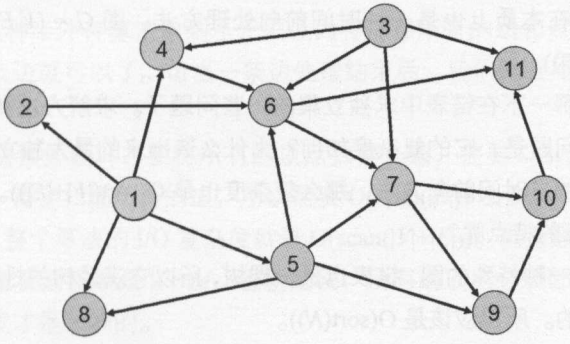
Mr. 王：这时我们就需要设计特殊的方法来解决，以下面的图为例进行说明。

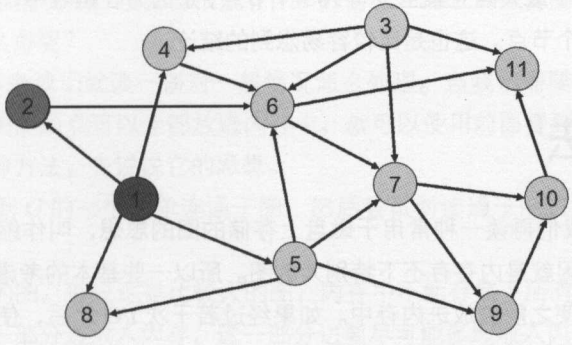


上面图中的每一个节点都有 ID。在原图中，边是没有方向的，为了能够使用时间前向方法，我们要将其转化为一个 DAG，也就是为这些边都加上方向。但是有一个原则，那就是所有的边都要从 ID 小的节点指向 ID 大的节点。

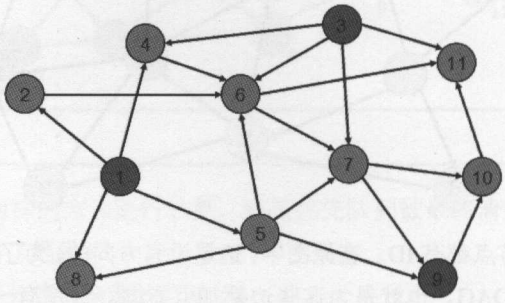
有了 DAG，我们就可以按照前面介绍的时间前向方法进行处理了。

对于这个图而言，首先我们访问序号最小的 1 号节点。它的后继节点是一定不会被加入到独立集中的，2,8,4,5 这几个节点是 1 的后继节点，等到访问它们时，记住不要将其加入到独立集中。根据拓扑序列，现在该访问 2 号节点了，但由于它是 1 的后续节点，所以不将其加入到 I 中。





在完成了对 1 和 2 号节点的访问之后，我们访问 3 号节点。3 号节点在访问 1 号节点时没有被标注为后继节点，所以 3 号节点可以被加入到 I 中。注意，2 的后继节点是不受影响的，因为 2 号节点并没有被加入到 I 中。当访问 3 号节点时，我们要记住 4,6,7,11 这几个节点是不能被加入到 I 中的。依此类推，直到访问了所有的节点。



小可在纸上写画了一会儿，说：对于这个图来说，好像最大独立集可以比这个大。

Mr. 王：没错。由于我们使用的是一个贪心算法，它求出的解不是最优解，但对于一个较大的图来说，这个最大独立集往往还可以接受，不是那么坏。

这是一个典型的时间前向处理问题，因为它具有时间前向的特点——每当访问一个节点时，它的父节点必然已经被访问过了，也符合拓扑排序的思想。而且它的时间复杂度也和时间前向处理方法相一致，它在本质上也是一种时间前向处理方法，图 $G = (V,E)$ 的最大独立集的 I/O 复杂度为 O(sort(|V|+|E|))。

现在我们可以解释一下在链表中求独立集的一些问题了。求解方法和求一般图的最大独立集是一样的。现在的问题是，它的复杂度如何？为什么选出来的最大独立集的节点有 $N/3$ 个？

小可：既然使用的是时间前向方法，那么复杂度也是 O(sort(|V|+|E|))。

Mr. 王：结合链表的特点呢？

小可：嗯，链表是一种特殊的图，链表也是一棵树，所以它满足树的性质 $E=V-1$。这就是说，E 和 V 几乎是一样大的。所以应该是 O(sort(N))。

Mr. 王：没错。至于最大独立集至少有 N/3 个节点，是因为在链表中，我们每选择一个节点，就会放弃其相邻的两个节点。这也是比较容易想到的结论。

5.7 缩图法

Mr. 王：接下来我们再谈一种常用于磁盘上存储的图的思想，叫作缩图法。我们不得不设计磁盘算法，重要原因就是内存存不下特别大的图。所以一些基本的考虑就是，我们能不能试着把图变得小一点，使之能被放进内存中。如果经过若干次 I/O 之后，使得图剩下的部分可以

被放进内存中，那么处理也就变得容易多了。

小可：哦，“缩图法”名字听起来也好像是这个思想。具体怎么做呢？

Mr. 王：我们来看这样一个问题：判定一个特别大的图的连通性。显然这个大图会被存储在外存中。

Mr. 王：首先我们试着给出一个半外存算法。

小可：这个“半外存”怎么解释呢？

Mr. 王：整个图的所有边和点可能是内存存不下的，但是如果图中的所有顶点都可以存放在内存中，那么对我们的处理将会相当有利。这在实际情况中还是比较常见的，因为对于比较稠密的图来说，边要比顶点多得多。

半外存的连通性判定是这样做的：首先，内存可以存得下所有的顶点，我们先将所有的顶点都放进内存中。接下来，我们从磁盘中读取边，逐条地把边放到内存中。每读取一条边，都做一个处理，就是将这条边删掉，然后将此边连接的两个顶点合成为一个顶点。我们之所以可以这么做，是因为图的连通性不会因为这种操作而被破坏，原来连通的部分依然连通，不连通的地方也就相应地没有边了。

小可：哦，这样不断地做下去，一个连通分量内的点不久就会被缩成一个点。最后当所有的边都被处理之后，如果内存中剩下的是一个点，则说明整个图都是连通的；否则就不是连通的。这个方法真巧妙。

Mr. 王：是的，这种半外存算法做到了无须让内存保存所有的边和点，内存只需要保存所有的顶点和图上的一条边就可以了。每当一条边处理结束后，我们就丢掉它。现在来看看算法的复杂度如何。

小可：首先，算法预处理时，要将所有的顶点扫描一遍，需要 O(scan(|V|)) 次 I/O。然后，在算法执行的过程中，需要扫描所有的边，所以需要 O(scan(|E|)) 次 I/O。

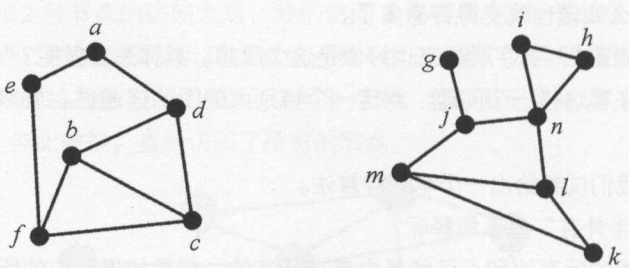
Mr. 王：嗯，所以整个算法的 I/O 复杂度就是 O(scan(|V|+|E|))。但是这里需要注意，可以采用这种半外存算法的重要前提就是 |V|<M。也就是说，所有的顶点必须都能放进内存中，这种做法及其相应的复杂度才是正确的。

小可：不过在实际情况中，也会有很多 |V|>M 的情况，对于那些真的大到连顶点都不能全部放进内存中的图怎么办呢？

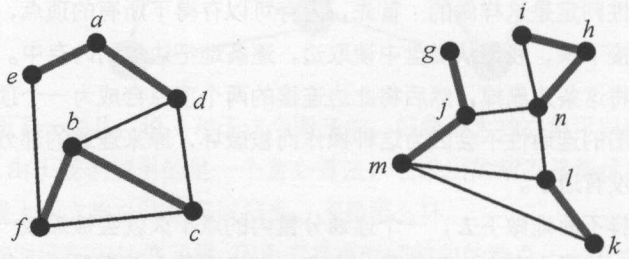
Mr. 王：嗯，接下来我们就谈一谈对一般情况怎么处理。当我们需要处理一个图时，首先要做一个判定：如果图的顶点可以全部放进内存中，就可以使用前面提到的半外存算法；如果不能，就尝试下面这种方法，先说说它的思想。

首先我们期望找到 G 的一个简单连通子图，然后尝试对连通子图进行收缩，使得这个子图的节点数变少。

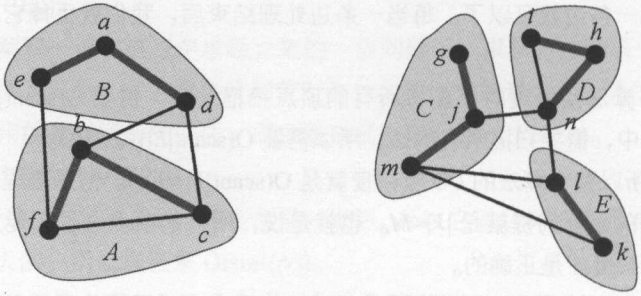
这是一般情况下的图。假设这是比较大的图，内存中不能存储下所有的节点。我们为了进行处理，只能先选取一部分边放入内存，这一部分边要尽可能多地放入内存中。



假设放入内存中的这些点就是所有的点，然后执行前面的半外存算法，尽可能多地将边装入内存中。



这时我们就能有效地发现一些连通分量。下一步是进行缩图，将这些边从内存中剔除；相应地，对内存中已经存在的连通分量进行合并，合并成一个新的点。



接下来进行下一轮迭代，将已经合并完的点当作新的顶点，继续将外存中的边逐步加入到内存中。

小可：此时顶点的命名已经发生了改变，是不是还要记住这些合并顶点间的对应关系呢？

Mr. 王：没错，比如新的顶点 A 和 B 之间的边，外存中保存的数据中并没有 A 和 B 顶点间的边，只有 ef 和 dc 这样的边，所以还要有机制来记住 e、f 这两个顶点之间的边，在下一轮迭代中，是 A、B 两个顶点之间的边。

持续地迭代下去，就可以求出结果了。下面我们来总结一下，在一般情况下，图的连通性判定是怎么做的。

第 1 步：对于每一个点找到其最小的邻居，如果图是使用邻接表来表示的话，就非常的容易，直接搜索其邻接表就可以了。

第 2 步：计算图 H 由选定的边导出的连通分量，这是我们前面描述算法的过程中主要完成的一项工作。

第 3 步：将每一个连通分量缩为一个节点，这也是非常容易的。

然后递归调用步骤 1~3，就可以完成图连通性的判定。

不过要记住，对于每一个 $v \in G'$，我们还要知道其在原图 G 中是由哪些节点合并来的，并且将其还原回去，将其连通性复制到其代表的图 G 中的每个节点上。这个也是比较容易实现的。

小可：那么这个算法的复杂度如何呢？

Mr. 王：现在我们一步一步来分析。首先，每加入一条边，都会构成一个新的连通分量，或者在已有的连通分量上增加一个点，这意味着每一个强连通分量的大小至少为 2。由此可知，每一次合并之后产生的图中点的数量和原图中点的数量具有这样的关系：$|v'| \leqslant |v|/2$。而我们最终期待的就是能将所有的点都放进内存中，所以只需要让原图中点的数量去比内存的大小，这个比是 $|v|/M$。每一次可以将 v 缩小一半，只要知道缩小多少次就可以将 v 缩小到 M 的大小即可，所以我们要执行递归调用 $\log_2(|v|/M)$ 次。

而在每一次递归调用中，我们要做的就是合并那些处在一个连通分量中的点。由于每一次都要寻找和某个点在邻接表中 ID 相邻的那些点与之形成的边，所以在进行合并时，相当于对边进行了一个排序，其复杂度为 sort(E)。

因此，计算出图 $G=(V,E)$ 的连通分量的 I/O 复杂度为 $O\left(\text{sort}(|E|)\log\left(|V|/M\right)\right)$。

小可：这个算法除了被用在判定连通性上，还可以被用来做什么呢？

Mr. 王：其实对图算法非常熟悉的人很快就可以想到，与连通性关联非常紧密的一类问题就是求解最小生成树。

小可：嗯，我知道求解最小生成树的经典算法有 Kruskal 算法和 Prim 算法，求解最小生成树也可以利用我们判定图连通性的框架进行吗？

Mr. 王：只需稍作改动。前面我们在寻找一个节点的临边时，采用的策略就是寻找 ID 和所选择的这个节点的 ID 最接近的顶点；而在求解最小生成树的过程中，我们不再选择 ID 最小的邻居，而是选择权重最小的边。然后在进行缩图时，压缩后的图中某条边的权值等于该边代表的所有边的权值的最小值。其实这就相当于将两个连通分量用其之间权重最小的那条边连接起来了。

小可：这个思想好像比较接近前面讲过的 Kruskal 算法。

Mr. 王：非常好，当向树中加入每一条将要被合并的边时，实际上加入了可以连接两个连通分量的最小边，而这同时也保障了不会出现两条连通分量被两条边连着。这是因为我们取过最小的那条边之后，两个连通分量就合为一个，不会再次被合并了。

这个算法的时间复杂度和前面的判断连通性相类似，我们还是尝试将整个图缩小到大小为 M，所以要经过$\log_2(|v|/M)$次迭代。在每一次迭代中，我们要去找连接两个连通分量的最小边，要对边集合 E 进行排序。综合起来，I/O 复杂度也是$O\left(\text{sort}(|E|)\log\left(\frac{|v|}{M}\right)\right)$。

下面总结一下所提到的图算法给我们的一些启发。

时间前向处理技术的思想是，将图问题转化为有向无回路图的估值问题，然后我们就可以利用最经典的表排序方法或者是 DAG 估值方法进行求解了。

缩图法正如其所言，在保持不损失我们求解问题所需要的特定信息的情况下，通过不断地迭代缩减图 G 的规模。在此过程中，在压缩图中递归解决问题。最后我们根据压缩图的解，构造图 G 的解。

缩图法和时间前向处理都包含着我们在解决数学或者计算机问题时一个很重要的思想或者说手段——“转化”。将难解问题或者未知问题等效转化为一个简单问题或者已知问题去求解，这是一种非常重要的能力。

另外，我们的算法中还隐含着一个思想叫作“自举”，一旦输入（或其部分）规模足够小，即可使用内存算法来求解。如果输入比较大，则可以通过不断地迭代和压缩，逐渐减小其输入规模，一旦发现下一轮迭代的输入规模已经小到可以放进内存中了，那么接下来的工作就可以交给内存算法去完成。相比磁盘 I/O，内存的存取速度是非常快的。

第6章 1+1>2——并行算法

6.1 MapReduce初探

Mr. 王：今天我们来谈一个新的话题——并行算法。

小可：并行？并行是不是说，一个任务由多个人同时做呢？

Mr. 王：通俗地讲是这样的。有很多问题，当数据规模比较大时，如果单独由一台计算机来做，就会变得费时费力，我们希望可以将一个问题交由多台计算机进行处理和解决。这就是我们要研究的并行算法。

小可：那具体要怎么做呢？如果把整个任务分开给多台计算机来做，我们就要想办法把任务分割开，还要对它们提交的结果进行综合，这对于一些复杂的问题还是有一定难度的啊。

Mr. 王：不要担心，我们有MapReduce。

小可：那是什么？

Mr. 王：MapReduce是一个分布式编程模型，最初是由Google公司的Jeffery Dean和Sanjay Ghemawat于2004年开发的。现在市面上流行的一个开源版本叫作Hadoop。这个编程模型让不精通于分布式并行开发的计算机工作者和程序员，也能有效、方便、快捷地开发并行程序。这样，即使是不了解并行编程的程序员，也可以用MapReduce将自己的程序并行运行在多台计算机上，实现并行计算。

顾名思义，MapReduce实现了两个功能：Map和Reduce。

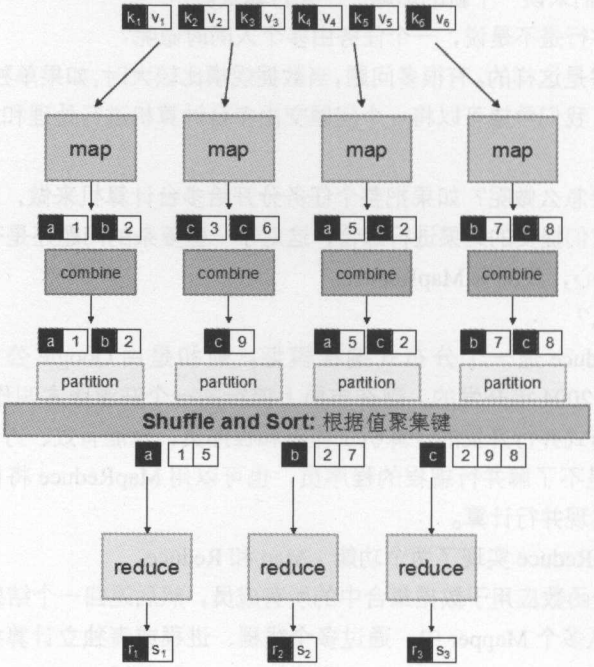
- Map是将一个函数应用于数据集合中的所有成员，然后返回一个结果集合。
- Reduce是把从多个Mapper中，通过多个线程、进程或者独立计算机系统并行执行的结果进行分类和归纳。

在使用 MapReduce 设计并行算法的过程中，程序员首先要定义 Map 函数和 Reduce 函数，将需要求解的问题用 Map 和 Reduce 这两种操作来描述。定义好之后，MapReduce 框架就可以帮助解决问题了。

小可挠挠头，说：还是很抽象啊。

Mr. 王：我们还是举一个例子来说明吧。比如统计一篇文章中各个字母出现的数量，这在破解替换密码中是一个非常重要的手段和步骤。所谓替换密码，就是用一个字母或者符号去替换另一个字母或者符号，比如用 x 来表示 e，用 a 来表示 t 等。因为在英文中，某个字母在一篇文章中出现的次数往往呈现一种分布。即使在信息的传递中，选择了另一个字母来替代这个字母，也是可以通过这个字母在大量文章中的统计百分比来判断它是哪一个字母的。假设现在需要破译一篇文章，我们猜测它使用了基于替换的密码，就需要统计每一个符号在整篇文章中的数量，比如经过统计，发现 x 出现的次数是最多的，根据以往的经验和大量的统计结论，密码专家就会猜测 x 这个字母可能表示的是 e。

现在我们用一个结构图来表示 MapReduce 在解决这个问题时，每一部分都做了哪些工作。整个算法的输入以 key-value 对的形式体现，由于统计字母数量这个问题比较简单，输入的单词仅有键（也就是字母）就可以了，在这个图中 Map 函数解决的问题，就是对 key-value 对中出现的字母进行一个初步统计。每当一个 Map 完成对长文章的一部分进行统计时，其将每个字母与其对应的出现次数组成 key-value 对，比如 <a,1><b,2> 等。



小可：第一个 Map 统计出的 a 有 1 个、b 有 2 个，可是第二个 Map 统计出的 c 分成了两个部分，怎么办呢？

Mr. 王：在实际的统计中，中间结果分块的情况非常常见，所以这里引入了一种机制，叫作 combine。它的引入就解决了统计结果分块的情况，注意看第二个 Map，它就将 c 的分块结果合并为 9。这样的做法非常有利于提高整个系统的效率。在这个例子中，如果相同的字母都被有效地合并，在最终进行统计时就会变得更加方便，否则后面的步骤就会变得更加麻烦。

接下来数据会经历一个叫作 partition 的过程，这个过程要做的是决定哪一个 key 给哪一个 Reducer 去处理。MapReduce 的好处之一就是，combine 和 partition 操作是可以由系统自动实现的，这两步是可选的，可以不用程序员去实现。

接下来系统自动执行 Shuffle 和 Sort，我们称之为洗牌和排序。此时 MapReduce 平台会将键值相同的数据项目洗混到一起，最后将每个键值的数据交给一个 Reducer 去处理。比如在这个例子中，Reduce 操作就是将 Reducer 接收到的 < 字母 , 出现次数 > 键值对进行合并，也就是将相同的字母对应的出现次数加起来，作为统计的结果。

简单整理一下，我们在设计一个 MapReduce 程序的主要工作时，就是确定 Mapper 需要执行何种 Map 操作，Reducer 执行何种 Reduce 操作。combine 和 partition 是可以由 MapReduce 平台框架自动去实现的，但是自动实现的效率往往不是最高的，如果程序员希望提高 MapReduce 平台在解决某些问题时的效率，则可以去自行定义 combine 和 partition 操作。

其中 combine 操作类似于一个微型的 Reducer，在 Map 执行过之后，combine 对 Map 的结果进行一个初步 Reduce。由于它进行的是一个合并操作，所以可以将具有相同键值的记录合并为一个，一大好处是减小了各台计算机之间的网络流量。比如我们发送 c=1、c=3、c=4 这样三条记录，会产生三条记录的流量；而如果发送一条 c=8 记录，那么只产生一条记录的流量，而且这不会影响最终的计数结果，因为它们在最后的 Reducer 处也是要合并的。partition 操作执行于 Reduce 之前，为不同的 Reducer 去分配其需要处理的键值范围。至于其他的各种操作，MapReduce 框架可以帮助程序员去完成一切。

小可：那这个“一切”都包括些什么呢？

Mr. 王：首先是 MapReduce 要进行调度工作，也就是为 Map 和 Reduce 这些操作分配“工人”。

小可：工人？

Mr. 王：也有些书籍称之为 Slaves，也就是去执行具体操作的那些计算机。因为在定义 MapReduce 时和 MapReduce 的具体运行过程中，我们并不知道 Map 和 Reduce 这些函数究竟具体运行在哪一台计算机上，Mapper 或 Reducer 何时启动、何时结束，一个特定的 Mapper 正在处理哪种输入，一个特定的 Reducer 正在处理哪个特定的中间键值。对于这些没有指定的工作都需要由 MapReduce 来执行，这样可以极大地减轻程序员管理大批计算机的辛苦。

其次是“数据分布”，进行将计算移动到数据的工作。“同步”完成聚集、排序、打乱中间数据的工作。当然不能忘了“错误处理”的工作，由于参加并行运算的计算机是很多的，中间

会涉及大量的网络通信，如果有“工人”出现“失败”、死锁、宕机的情况，或者中间网络通信流量出现拥堵，MapReduce 平台要进行报警、重新启动或尝试恢复这些机器，使其正常运转。

在 MapReduce 算法的设计过程中，我们的最重要工作就是对算法用 Map 和 Reduce 来进行描述，有时还需要 combine 操作。

Combine 体现了本地聚合的思想。在前面的例子中也提到了，我们可以在进行 Mapper 发出数据之前，将其在本地进行提前聚合，目的是为了减小通信量，同时也让 Reduce 的过程变得更加轻松。比如统计结果包括 c=2 和 c=3 两条记录，如果 Mapper 将其都发送出去，通信量就是两条记录；如果在本地将其相加为 c=5 的话，那么只需要发送一条记录即可。

在使用并行系统时，由于涉及很多计算机之间的通信，而通信往往是多机系统的效率瓶颈之一所以我们应尽可能多地让数据在本地计算、本地合并、传输结果，而不是将未经处理的数据一一发送出去。

6.2 MapReduce 算法实例

Mr. 王：我们来看几个 MapReduce 应用的实际例子，这样更有助于你对它的认识。

小可：我也迫不及待地想试试 MapReduce 的应用了。

6.2.1 字数统计

Mr. 王：先讲一个最基本的应用——字数统计。这个例子与前面的字母计数是非常相似的，只不过这里要统计的是单词数目。现在我们来更具体地说说，它的 Map 和 Reduce 是如何进行设计的。

Mr. 王拿出一块白板，在上面写下了一段代码，说：这里有一段代码，它实现的就是字数统计的 Map 和 Reduce，它并不复杂。

```
1: class Mapper
2:     method Map(docid a, doc d)
3:         for all term t ∈ doc d do
4:             Emit(term t, count 1)
1: class Reducer
2:     method Reduce(term t, counts [c1, c2, ...])
3:         sum ← 0
4:         for all count c ∈ counts [c1, c2, ...] do
5:             sum ← sum + c
6:         Emit(term t, count s)
```

可以看出，这个 Map 函数的输入是文档编号和相应的文档；Reduce 函数的输入是文档中出现的某一个单词和其频度统计。在这个算法中，Map 和 Reduce 的执行逻辑其实也是很简单的。Map 函数将一个文档打开，每遇到一个单词，就发送 < 单词 ,1> 这样的一个元组。也就是说，每当 Map 函数发现了某个单词时，它就报告自己发现的这个单词是什么，同时希望 Reduce 函数在这个单词的计数上加 1。而 Reduce 函数将整理来自所有 Map 的结果，并将相同的单词对应的这些“1”进行累和，输出该单词和统计频度。

你来说说，这个算法的缺陷在哪里？

小可看了看面前的白板，说：我觉得这个 Map 函数太简单了，发现一个单词就传输一个 1 出去，这样就会有大量的时间都消耗在了传输这些 1 上面，每有一个单词就要传输一次，效率太差了。

Mr. 王：应怎么改进？

小可：能不能让 Mapper 在内部对某个单词的频度进行一个统计，使其可以存储一些中间结果，而不是急着把它们发送出去呢？

Mr. 王开始擦除白板上的算法，说：我们来看一个改进版本。

```
1: class Mapper
2:     method Map(docid a, doc d)
3:         H ← new AssociativeArray
4:         for all term t ∈ doc d do
5:             H{t} ← H{t} + 1                ▷ Tally counts for entire document
6:         for all term t ∈ H do
7:             Emit(term t, count H{t})
```

小可：这个改进版本多了一个数组 H，可以对单词 t 进行内部统计，在输出结果之前，将相同的单词在这篇文章里出现的次数进行了合并，原来是出现 100 次单词就要发送 100 条记录，现在只需要发送 1 条记录就可以了，这样的确极大地减小了在网络中传输的数据量。

Mr. 王：这里有一个问题，当我们采用了这个改进版本的 Mapper 时，是不是还需要使用 combiner 呢？

小可：我觉得不需要了吧，统计工作已经由 Mapper 做了。

Mr. 王：不对，combine 依然是需要的。我们注意看这个版本 Mapper 的描述，它中间有一句是“for all term t ∈ doc d”，这意味着它运行时虽然会将来自一个文档的相同词汇及时合并，但是一个 Mapper 可能会去处理很多个文档，而我们定义的这个 Mapper 是不能将来自多个文档的相同词汇进行合并的，比如在 doc1 中，我们发现单词 t 出现了 100 次，在 doc2 中出现了 200 次，这个 Mapper 就会发出 2 条记录。不难发现，对于一个单词 t 最终还是发出了多条记录，当文档非常多时，发出的记录也就非常多，这对于通信来说依然会造成很大的压力。

所以我们还要继续改进：

```
1: class Mapper
2:     method Initialize
3:         H ← new AssociativeArray
4:     method Map(docid a, doc d)
5:         for all term t ∈ doc d do
6:             H{t} ← H{t} + 1                    ▷ Tally counts across documents
7:     method Close
8:         for all term t ∈ H do
9:             Emit(term t, count H{t})
```

Mr. 王：我们来看看这个最终版本，这里有什么不同呢？Mapper 把数组声明成了一个全局的量，Mapper 在其整个运行过程中，不论它在处理哪一个文档，都能访问到这个数组。也就是说，在 Mapper 运行的过程中，只要对应的是一个单词，不论它来自哪一个文档，都可以将其加入到单词在数组中的频度记录中去，这就实现了 Mapper 在其处理的所有文档中对单词统计量的合并。最后，当 close 函数被执行时，Mapper 会将整理好的数据发送出去。

小可：那么这个时候，是不是可以彻底地取消 combiner 了呢？

Mr. 王：是的。此时 combine 操作的所有工作都已经被 Mapper 实现了，能聚集的已经被 Mapper 都聚集了。这种设计模式叫作本地聚合，或者叫作 In-Mapper。这种做法就是希望将 combine 操作都集成到 Mapper 中去，它保持了多个 Mapper 调用中的状态。这样做的好处就是，如果 Mapper 不去做这种 In-Mapper 化处理，而是直接将大量的 <term,1> 输出出去，指望着 combiner 去完成聚合工作，那么每一个 Mapper 就会有大量的数据输出。可以想象，这个数据量会非常大，是很容易超过内存缓冲区大小的，一旦内存空间被占满，数据就不得不被缓冲到磁盘上，当 combiner 要调用这部分内容时，就要从磁盘上把这部分缓冲数据取出来，这就造成了两次磁盘 I/O。我们也知道，磁盘 I/O 的速度相比内存读写慢得太多了，这是非常影响系统运行效率的。通过引入这种 In-Mapper 模式，可以有效地将工作转变为内存操作，相比初始的版本效率高了很多。

小可：不过设计一个好的 In-Mapper 也是很需要设计者经过一番深入思考的啊。

Mr. 王：的确，单词数目统计这个问题还算是非常简单的，对于一些比较复杂的问题，就需要程序员有比较清晰的思路和良好的设计。另外，本地聚合还需要比较大的内存空间，这意味着程序员要进行显式的内存管理，对系统进行优化，这对程序员的编程功底也是一个考验。

6.2.2 平均数计算

Mr. 王：再来看一个例子——均数计算。我希望借助这个例子，仔细讲解一下关于 combiner 的问题。

小可：从前面的例子可以看出，其实 combiner 和 Reducer 挺像的，它们做的都是合并工作。

Mr. 王：没错。它们的确有很多相似之处。

小可：那直接把 Reducer 拿出来做 combiner 就好了啊。

Mr. 王：有的时候的确可以这样实现，但是绝大多数时候不行。至于为什么不行，我会在后面告诉你。但是需要记住的一点是，combiner 是一个可选的优化，不论有没有 combiner，程序都必须能正确地运行出结果；而 combiner 的出现，只是提高了系统运行效率。combiner 可能运行，也可能不运行，还可能会运行多次，这与具体的数据项构成有关。

好，回到例子上，这个例子是找到与相同键值相关联的所有整数的平均数。

Mr. 王转身拿出了白板，把算法写在了上面。

```
1: class Mapper
2:     method Map(string t, integer r)
3:         Emit(string t, integer r)

1: class Reducer
2:     method Reduce(string t, integers [r_1, r_2, ...])
3:         sum ← 0
4:         cnt ← 0
5:         for all integer r ∈ integers [r_1, r_2, ...] do
6:             sum ← sum + r
7:             cnt ← cnt + 1
8:         r_avg ← sum/cnt
9:         Emit(string t, integer r_avg)
```

Mr. 王：这个是版本 1。你来解释一下这个算法的工作过程。

小可指着白板，说：这个 Mapper 几乎什么都没做啊，遇到一个字符串和整数对，就将其传递给 Reducer。至于 Reducer，它是根据字符串进行匹配的，将具有相同键值的字符串以及对应的整数值收集到一起，然后剩下的部分就是对这些值求平均数，sum 累计所有的整数 r，cnt 对其出现的 r 的数量进行计数，最后返回它。

Mr. 王：想一想，这里的 Reducer 能不能用来做 combiner？

小可想了想，说：这里的 Reducer 做的就是平均数计算，如果把它用作 combiner 的话，中间就会产生很多只带有平均数值的结果。

Mr. 王：用这样的结果，能求出最终的平均数吗？

小可：平均数的算术平均数不是所有值的平均数，所以结果不对。

Mr. 王：好，那我们来看看版本 2。

```
1: class Mapper
2:     method Map(string t, integer r)
3:         Emit(string t, integer r)

1: class Combiner
```

```
2:    method COMBINE(string t, integers [r1, r2, ...])
3:        sum ← 0
4:        cnt ← 0
5:        for all integer r ∈ integers [r1, r2, ...] do
6:            sum ← sum + r
7:            cnt ← cnt + 1
8:        EMIT(string t, pair (sum, cnt))              ▷ Separate sum and count
1: class REDUCER
2:    method REDUCE(string t, pairs [(s1, c1), (s2, c2) ...])
3:        sum ← 0
4:        cnt ← 0
5:        for all pair (s, c) ∈ pairs [(s1, c1), (s2, c2) ...] do
6:            sum ← sum + s
7:            cnt ← cnt + c
8:        ravg ← sum/cnt
9:        EMIT(string t, integer ravg)
```

小可：这个版本的 combiner 携带了每个平均数的 count，我们可以通过这个 count 来还原每一组平均数的总数，最后通过 count 的和与每一组平均数的和来求出所有数据的平均数，这样就能在 Reducer 中求解出总的平均数了。这个版本是比较不错的。

Mr. 王：此言差矣，这个版本是不能用的。

小可一脸惊讶地说：这是为什么呢？看起来是一种很不错的设计啊。

Mr. 王：想一想，combiner 对于程序来说是不是一个必要的环节？

小可：不是，combiner 有助于优化效率，但是去掉它也不影响 MapReduce 的运行。

Mr. 王：那么在这个版本里面呢？

小可恍然大悟，说：哦，这个版本的确有问题，combiner 不仅进行了优化，而且还改变了输入输出数据类型，如果在这里去掉 combiner 的话，那么 Mapper 函数的输出数据类型与 Reducer 函数的输入数据类型就不匹配了。

Mr. 王：在设计和编写程序时一定要细心。这里我们在 Mapper 里面稍作修改，请见版本 3。

```
1: class MAPPER
2:    method MAP(string t, integer r)
3:        EMIT(string t, pair (r, 1))
1: class COMBINER
2:    method COMBINE(string t, pairs [(s1, c1), (s2, c2) ...])
3:        sum ← 0
4:        cnt ← 0
5:        for all pair (s, c) ∈ pairs [(s1, c1), (s2, c2) ...] do
6:            sum ← sum + s
7:            cnt ← cnt + c
```

```
8:        EMIT(string t, pair (sum, cnt))
1: class REDUCER
2:    method REDUCE(string t, pairs [(s1, c1), (s2, c2) ...])
3:        sum ← 0
4:        cnt ← 0
5:        for all pair (s, c) ∈ pairs [(s1, c1), (s2, c2) ...] do
6:            sum ← sum + s
7:            cnt ← cnt + c
8:        r_avg ← sum/cnt
9:        EMIT(string t, pair (r_avg, cnt))
```

小可：嗯，这次即使把 combiner 彻底去掉，也不会影响整个程序的运行结果，只是在 Mapper 上面稍作修改，效果还是很好的。

Mr. 王：想想看，还可以怎么改?

小可：前面提到过一种设计思想叫作 In-Mapper，对于这种需要额外定义 combiner 的，可以试试吧。

Mr. 王：很好，我们给出版本 4。

```
1: class MAPPER
2:    method INITIALIZE
3:        S ← new ASSOCIATIVEARRAY
4:        C ← new ASSOCIATIVEARRAY
5:    method MAP(string t, integer r)
6:        S{t} ← S{t} + r
7:        C{t} ← C{t} + 1
8:    method CLOSE
9:        for all term t ∈ S do
10:           EMIT(term t, pair (S{t}, C{t}))
```

这个版本把 combiner 做的事情全部集成到了 Mapper 中，使得 combine 操作在执行 Map 函数时就做到了，进一步减小了程序的通信复杂度。

6.2.3　单词共现矩阵计算

Mr. 王：这里还有一个很典型的例子——单词共现矩阵计算。

这个例子是计算文本集合中词的共现矩阵。我们设 $\boldsymbol{M}$ 是一个 $N \times N$ 的矩阵，其中 N 为词数，矩阵中的 M_{ij} 表示 i 和 j 在同一个上下文中的次数。

小可：这个上下文是什么呢?

Mr. 王：上下文可以是一个句子，也可以是一个段落，这要视实际情况而定。

小可：那么单词共现矩阵计算有什么用呢？

Mr. 王：这是一种用来测量语义距离的方法。两个词出现在同一个句子中的次数越多，说明它们之间的语义距离就越近，它们之间的关联性也就越大。“语义距离”这个量，在很多的自然语言处理任务中发挥着很重要的作用。

这是一个典型的大规模计数问题，它具有大规模计数问题的几个主要特征。首先，它有一个大的事件空间（单词数目）；其次，它会产生大量的观测值（单词集合）。而我们的目标是记录关于事件的统计数据。

小可：具体应该怎么做呢？

Mr. 王：解决这类问题的一个基本方法，就是让 Mapper 来生成对多个文档的部分计数，Reducer 对部分计数进行聚合。

小可：这和前面我们使用的方法也是十分类似的。

Mr. 王：没错，但是现在我们面对的核心问题就是，如何高效地对部分计数进行聚合。我们首先可以想到的基本方法就是词对法。当 Mapper 处理一个句子时，生成这个句子里面的共现词对。对于所有的词对，Mapper 会发出一个 (*a*,*b*) 为 key、计数为 value 的键值对，Reducer 将这些来自 Mapper 的词对 + 计数键值对进行聚合，得出最终结果。下面是算法的伪代码。

```
1: class Mapper
2:     method Map(docid a, doc d)
3:         for all term w ∈ doc d do
4:             for all term u ∈ Neighbors(w) do
5:                 Emit(pair (w, u), count 1)        ▷ Emit count for each co-occurrence
1: class Reducer
2:     method Reduce(pair p, counts [c1, c2, ...])
3:         s ← 0
4:         for all count c ∈ counts [c1, c2, ...] do
5:             s ← s + c                              ▷ Sum co-occurrence counts
6:         Emit(pair p, count s)
```

不难看出，在 Mapper 中，对于任何一个词 *w*，将它所有的邻居 *u* 与之组成一个 pair(*u*,*w*)，于是每一个 (*u*,*w*) 都拥有计数 1，我们将 ((*w*,*u*),1) 这样的键值对发送出去。在 Reducer 中，对于每一个 pair *p* 和来自 Mapper 的各种计数累和，最后返回 (*p*,count) 这样的键值对，就成功地实现了单词贡献矩阵计算。

小可：不过这样做的通信复杂度就会很大吧。

Mr. 王：没错，这种做法虽然易于实现，但其洗牌和排序的复杂度会非常大，效率真的很差。你想它的 key 都是一些词对，这意味着 key 的取值空间是非常大的。

小可：那么我们在 Mapper 后面加上一个 combiner 会不会好一些呢？

Mr. 王：增加 combiner 确实是一个比较常用的方法，但是在这个问题中，combiner 却很难

发挥它的作用，原因还是 key 的空间过大。key 用 (a,b) 这样的形式来表示，就意味着 (a,b)、(a,c) 等都不是相同的 key，本身在一个 Mapper 中，相同的键值对就非常少，可以进行聚集的键值对就不多。所以用 combiner 这种策略对于改进这个算法的效果不够好。

小可：那还有什么更有效的策略吗？

Mr. 王：这里介绍一种方法，叫作条带法。前面引起很大困难的原因是键值设计过于复杂，其空间太大导致了排序和洗牌的混乱。这次我们把 key 就设为单词。

原来，我们设定的词对是这样的：

$(a,b) \rightarrow 1$

$(a,b) \rightarrow 2$

$(a,b) \rightarrow 5$

$(a,b) \rightarrow 3$

$(a,b) \rightarrow 2$

现在调整为：$a \rightarrow \{ b: 1, c: 2, d: 5, e: 3, f: 2 \}$

我们记录与 a 共现的单词分别有哪些，它们出现的次数是多少，而不是记录共现对出现的次数。

小可：这样做，key 的取值空间就会大大减小。

Mr. 王：此时，当 Mapper 需要处理一个句子时，我们发出的键值对的形式就是：

$$\text{word}_a \rightarrow \{\text{word}_b:\text{count}_b, \text{word}_c:\text{count}_c, \ldots\}$$

到了 Reducer 之中，我们再将上述的键值对进行合并：

$$\begin{array}{l} a \rightarrow \{b:1 \quad d:5, e:3\} \\ +a \rightarrow \{b:1, c:2, d:2, \quad f:2\} \\ \hline a \rightarrow \{b:2, c:2, d:7, e:3, f:2\} \end{array}$$

但是这个问题的关键点在于，如何设计一个好的数据结构，让后面的 value 部分能够更容易聚合。

我们可以设计这样一个数组，该数组将每一个词映射成一个数组下标，然后当某个词 u 出现在词 w 的上下文中时，我们将其对应的下标在 w 对应的数组中的位置中的计数值加 1。最后发出的是 (Term w,Array H) 这样的键值对。

```
1: class Mapper
2:     method Map(docid a, doc d)
3:         for all term w ∈ doc d do
4:             H ← new AssociativeArray
5:             for all term u ∈ Neighbors(w) do
6:                 H{u} ← H{u} + 1                ▷ Tally words co-occurring with w
7:             Emit(Term w, Stripe H)
```

```
1: class Reducer
2:     method Reduce(term w, stripes [H1, H2, H3, ...])
3:         Hf ← new AssociativeArray
4:         for all stripe H ∈ stripes [H1, H2, H3, ...] do
5:             Sum(Hf, H)                          ▷ Element-wise sum
6:         Emit(term w, stripe Hf)
```

接下来在 Reducer 中，我们将关于 term w 的条带数组进行聚合，从而得出所需要的结果。

这个算法的优势在于，它的 key 空间相比前面的词对要小得多，这意味着它能够更好地利用 combiner。

但是这种做法实现起来相对会困难一些，而且这个算法里面潜在的对象是非常大的。我们为每一个词申请的数组，是造成潜在对象非常大的首要原因。

下面我们看看如何进一步应用所求出来的单词共现矩阵。在自然语言处理中，我们经常需要通过共现矩阵求出两个单词间的相对频率。其表达式是这样的：

$$f(B \mid A) = \frac{\text{count}(A,B)}{\text{count}(A)} = \frac{\text{count}(A,B)}{\sum_{B'} \text{count}(A,B')}$$

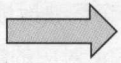
小可：这个 count(A,B) 就是词 A 和词 B 的共现计数吧？

Mr. 王：没错。现在需要思考的是，如何利用 MapReduce 来解决这个问题。首先来看看条带法。对于条带法，我们只要使用共现矩阵关于 A 的那个数组就可以了。扫描一遍，就知道 A 出现了多少次，在扫描第二遍时，就可以根据 count(A,B) ——求出对于任意一个 B，$f(B \mid A)$ 的值。

小可：那么对于词对法，又是怎么做的呢？

Mr. 王：对于词对法，要考虑的问题就比较多了。我们要对词 A 额外设计一个词对为（a,*）。

$(a, *) \rightarrow 32$	Reducer 在内存中保存该值



$(a, b1) \rightarrow 3$	$(a, b1) \rightarrow 3 / 32$
$(a, b2) \rightarrow 12$	$(a, b2) \rightarrow 12 / 32$
$(a, b3) \rightarrow 7$	$(a, b3) \rightarrow 7 / 32$
$(a, b4) \rightarrow 1$	$(a, b4) \rightarrow 1 / 32$
…	…

(a,*) 这个量保存的是 a 出现的总频数，也就是 count(A)。为了完成上面的工作，我们在设计 MapReduce 时，要额外考虑以下几个方面。

- Mapper 需要额外传递一个 (a,*) 用于求 count(a)，还要保证它必须先于其他所有的 (a,b)

先抵达 Reducer，这样计算才能有效地继续进行下去。

- 必须要保证所有的同一个词 *a* 都会传递给同一个 Reducer。
- 还要在涉及不同的 key-value 对的 Reducer 中保存相应的状态。

小可：各种方法在处理问题时还是各有利弊啊。

Mr. 王：在计算机科学中非常流行的一个词汇叫作 Trade-off。Trade-off，简而言之，就是在有限的计算资源制约下，资源之间的互相置换和平衡。常见的例子有空间换时间和时间换空间等。

小可：都是怎么解释的呢?

Mr. 王：比如我们要解决一个问题，当希望利用较小的内存空间去解决时，就需要花费较多的时间；当希望节省时间时，可能就要占用较大的内存空间。我们要寻找一种时间和空间都能接受的方法，这个过程就是 Trade-off。

在 MapReduce 的设计中，也涉及很多 Trade-off 的问题。比如键值对的数量控制，创建对象的数量越多，开销就越大，同时也会对排序和洗牌的效率造成一些影响；而如果减小键值对的数量，单个键值对的大小可能就会变得比较大，这意味着在传输过程中，同样会造成通信比较耗时的问题。另外，对于本地聚合问题，也是很值得思考的。对于不同的问题，combiner 的设计会产生很大的效率变化。当然，我们也可以尝试去设计 In-Mapper 和 combiner 共存的 MapReduce 程序。还有就是联系我们之前研究过的磁盘算法。在算法执行过程中产生的大量中间结果，是存到内存中还是磁盘上，或者是在整个机架、机群的网络中传输，都会产生非常不同的效果，这是一个好的 MapReduce 使用者或者说程序员不得不深入考究的问题。

好了，今天听了这么多，你也很累了吧，我们的课就上到这里，下次再见。

小可：好的，王老师再见。

6.3　MapReduce 进阶算法

Mr. 王：现在我们来谈些复杂点的 MapReduce 算法。

6.3.1　join 操作

Mr. 王：第一个话题就是 join 操作。join 操作在数据库中还是非常常见的。

小可：这个 join 指的是笛卡儿积操作吗?

Mr. 王：还不一样，但是我们要从笛卡儿积说起。

先来回顾一下笛卡儿积操作。在数据库中，数据的基本单位就是元组。比如在学生成绩的数据库中，表头是学号、学生姓名和成绩，那么元组就是 <0001, 张三 ,99>、<0002, 李四 ,90>

这样的。

如果元组为 $r=(r_1,\cdots,r_n)$、$s=(s_1,\cdots,s_m)$，则定义 r 与 s 的串接为：

$$rs=(r_1,\cdots,r_n,s_1,\cdots,s_m)$$

下面给出笛卡儿积的定义。

有两个关系 R、S，其度分别为 n，m，则它们的笛卡儿积是所有这样的元组集合：元组的前 n 个分量是 R 中的一个元组，后 m 个分量是 S 中的一个元组。

$R\times S$ 的度为 R 与 S 的度之和，$R\times S$ 的元组个数为 R 和 S 的元组个数的乘积。

而 $R\times S$ 这个集合为：

$$R\times S=\{\hat{rs}\mid r\in R\,\&\,s\in S\}$$

小可：嗯，笛卡儿积就是将 R 和 S 中的元组两两进行串接。

Mr. 王：笛卡儿积是各种连接操作的原型操作，如果给笛卡儿积操作添加各种限制条件，就是数据库中的各种连接操作 join。

连接的一种抽象表示叫作 θ-join。

$$R\underset{A\theta B}{\triangleright\triangleleft}S=\{\hat{rs}\mid r\in R\,\&\,s\in S\,\&\,r[A]\theta s[B]\}$$

θ 是一个算术比较符号，当 θ 是等号时，这个连接为等值连接。

R

A	B	C
1	2	3
4	5	6
7	8	9

S

D	E
3	1
6	2

$R\underset{B<D}{\triangleright\triangleleft}S$

A	B	C	D	E
1	2	3	3	1
1	2	3	6	2
4	5	6	6	2

比如在这个例子中，θ 就定义为 $B<D$，可以看到右边是连接结果。

这里还有一个很常用且很重要的连接，叫作自然连接。

$$R\triangleright\triangleleft S[\overline{B}]=\{\hat{rs}[\overline{B}]\mid r\in R\,\&\,s\in S\,\&\,r[B]=s[B]\}$$

自然连接是连接那些在相同的列上面有着相同的属性的元组。与等值连接的不同之处在于，自然连接要在结果中去掉重复的属性，而等值连接则不必。

r

A	B	C	D
α	1	α	a
β	2	γ	a
γ	4	β	b
α	1	γ	a
δ	2	β	b

s

B	D	E
1	a	α
3	a	β
1	a	γ
2	b	δ
3	b	$\in$

$r\triangleright\triangleleft s$

A	B	C	D	E
α	1	α	a	α
α	1	α	a	γ
α	1	γ	a	α
α	1	γ	a	γ
δ	2	β	b	δ

小可：嗯，在自然连接中，r 和 s 中都有的 B 属性和 D 属性在其结果中被去掉了。

Mr. 王：在数据库系统的各种操作中，连接的开销是非常大的，所需要的 CPU 资源和内存资源，甚至是保存中间结果的磁盘资源都是非常多的，于是我们产生了一个很自然的想法，就是做并行连接操作。其实在 MapReduce 出现之前，就已经出现了很多解决高开销的连接操作办法；在 MapReduce 出现之后，它也被用来解决并行连接问题。在这个方面也发表过很多篇论文，比如：

Map-Reduce-Merge: Simplified Relational Data Processing on Large Clusters SIGMOD 07

Semi-join Computation on Distributed File Systems Using Map-Reduce-Merge Model SAC10

Optimizing joins in a map-reduce environment VLDB09, EDBT2010

A Comparison of Join Algorithms for Log Processing in MapReduce SIGMOD 10

各种算法的实现思路如下表所示。常见的对 join 的实现有 Sort-Merge join、Hash join 等。Sort-Merge join 就是对两张表分别排序，然后利用类似归并排序的方法进行合并，如果来自两张表的两条记录存在相同的连接键值，那么进行连接。

Hash join 利用一种叫作散列表的查找结构，首先对第一张表建立查找结构（即散列表 T），然后用第二张表中各条记录的连接键在散列表中进行查找。关于散列表的实现细节，我们会在后面的内容中讨论。

Sort-Merger join	Map	区间 partitioner，生成排序的桶，每个桶对应一个 Reducer
	Reduce	从所有的 Mapper 中读取桶并且将其归并到一个排序的集合中
	Merge	从两个数据集合中读取排序的桶并执行 Sort-Merge join
Hash join	Map	Hash partitioner，桶哈希，每个桶对应一个 Reducer
	Reduce	从所有的 Mapper 中读取桶，使用 Hash 表分组和聚集这些记录（使用和 Mapper 相同的 Hash 函数），无须排序
	Merge	内存 Hash join
Block Nested loop join	Map	同 Hash join
	Reduce	同 Hash join
	Merge	Nested loop join

在本质上，这些算法是传统关系型数据库中连接算法的并行实现版本。

小可：在这些并行实现里面，除了 Map 和 Reduce，还多了一个 Merge。

Mr. 王：没错，这些算法在得到 MapReduce 执行结果之后，要额外经历一个 Merge 过程，才能得到最终的连接结果。

比如 Sort-Merge join，Mapper 对两个表分别进行一次区间 partition，生成排序的区间桶，每个桶对应一个 Reducer。然后 Reducer 会从 Mapper 中读取桶，这些桶是落在同一个区间内的数据，接下来进行归并，相当于把两个表分别进行了一次归并排序。最后 Merge 操作对这些排好序的桶执行 Sort merge join。

Hash join 与之类似，Mapper 对两个表分别进行一次 Hash partition，生成排序的 Hash 桶，每个桶对应一个 Reducer。Reducer 会从所有的 Mapper 中读取桶，使用 Hash 表分组和聚集这些记录，当然这要选取和 Mapper 一样的 Hash 函数，无须进行排序。等到了 Merge 阶段，只需要进行内存的 Hash join 就可以了。

小可摇了摇头，说：这一段没太听明白，都是一些很抽象的概念。

Mr. 王：那我们就来看看 Hash join 具体是怎么做的吧。

两个表直接拿过来，我们不对其做任何排序和预处理。对这两个表进行一些随机分割，然后 Mapper 会去读取这些分割好的表块，并将它们划分为 Hash 桶。最后这些 Hash 桶根据相应的 Hash 值归入相应的 Reducer 中。在 Reducer 中，将归入一个 Reducer 中的两个表的表块合并成一个表。于是每个 Reducer 的输入对应的就是相同的 Hash 值，因此就可以放到同一个 Merger 里面去进行合并，此时只要进行传统的内存 Hash join 就可以了。

另外还有一种方法，是在 MapReduce 的扩展模块 Hive 中的一种合并方法，叫作 repartition join。这是专门针对 MapReduce 提出的一种并行 join 方法。假设 L、R 分别为两个输入的表，我们将其规范化成 key-value 对。

小可：可是此时，什么是 key、什么是 value 呢?

Mr. 王：很自然的，我们将要做连接的键值作为 key，一般就是两个表中相同的属性。经过 Map，相当于将这些 key-value 对根据 key 进行了分组。通过洗牌，这些具有相同 Key 的元组就被分到了相同的组中，不管它是来自表 L 还是表 R。而在 Reducer 中，同一个 Reducer 收到的就是具有相同 key 的元组，可以直接根据表的名字做笛卡儿积。最后把结果组合起来输出就可以了。

小可：这个方法听起来不错，思路很清晰。

Mr. 王：不过这个方法也是有其缺点的。在算法的执行过程中，有可能需要缓存所有的记录，这可能导致内存溢出。

小可：那么现在有什么好的解决办法了吗?

Mr. 王：这里有一个解决方案，首先在 Map 部分，我们将输出键值设为连接键和表名的一个组合。在 Partition 函数里面，HashCode 仅从连接键进行计算。

可以看出，使用前面的办法已经可以基于 MapReduce 来完成表的并行 join 操作了。

Mr. 王：还有一个问题，我们前面一直讨论的是等值连接或者自然连接，而在很多应用背景中，都会提到一个叫作“相似连接”的概念。

小可：这个相似要怎么解释呢?

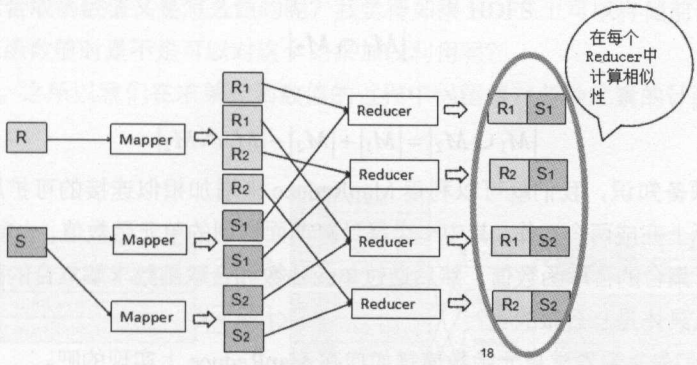
Mr. 王：比如你使用 Google 要查找某内容，但是输错了一个字母，Google 会提示你输入的是不是某个常用的候选值。这就是相似查询。在传统的关系型数据库查询中，自然连接或者等值连接都可以严格地通过判等进行连接，而当执行相似的或者模糊判等操作时，它的计算时

间一定会比精确连接更慢。于是我们定义了以下问题。

问题：一对来自两个数据集的记录，如果它们的相似性超过一定的程度，那么它们应该被连接，相似度可以根据特定应用来定义。

现在你能不能试着对这个问题给出一个朴素的解法呢?

小可：我觉得可以试着这样做，用 Mapper 将两个表分别 Map 到不同的键值上去，然后将它们的组合送到 Reducer 中，在 Reducer 中进行比较，我们观察它们的相似性是不是足够大，如果足够大就进行组合。



Mr. 王：这个做法虽然很简单，但是如果有 K 个块的话，我们就需要有 K^2 个 Reducer，这是十分消耗计算资源的。它的中转数据也是很大的，有 $K(R+S)$ 个。所以这个方法虽然正确，但是太耗时了。

在实际的计算中，我们可以根据表中记录所具有的一定性质，来使用一些更加聪明的办法，使问题的求解变得更加高效。

我们举一个多元相似的例子。

假设有两个集合 M_1 和 M_2。下面的表示法表示 M_1 中有 2 个 a、1 个 b、3 个 c；M_2 中有 1 个 a、2 个 c、2 个 d。

$$M_1=\{<a,2>,<b,1>,<c,3>\},M_2=\{<a,1>,<c,2>,<d,2>\}$$

衡量这样的集合的相似度有一些比较经典的判据，比如：

- Jaccard 相似度

$$\left|\frac{M_1\cap M_2}{M_1\cup M_2}\right|=\frac{|<a,\min(2,1)>,<c,\min(3,2)>|}{|<a,\max(2,1)>,<b,1>,<c,\max(3,2)>,<\mathrm{d},2>|}=\frac{1+2}{2+1+3+2}=\frac{3}{8}$$

- Cosine 相似度

$$\frac{M_1\cap M_2}{\sqrt{|M_1|-|M_2|}}=\frac{1-2}{\sqrt{(2-1-3)-(1-2-2)}}=\frac{3}{\sqrt{11}}$$

• Dice 相似度

$$2\times\frac{|M_1\cap M_2|}{|M_1|-|M_2|}=\frac{1-2}{\sqrt{(2-1-3)}-(1-2-2)}=\frac{6}{\sqrt{11}}$$

常见的计算也有如下几种。

• 单元函数

$$|M|$$

• 合取函数

$$|M_1\cap M_2|$$

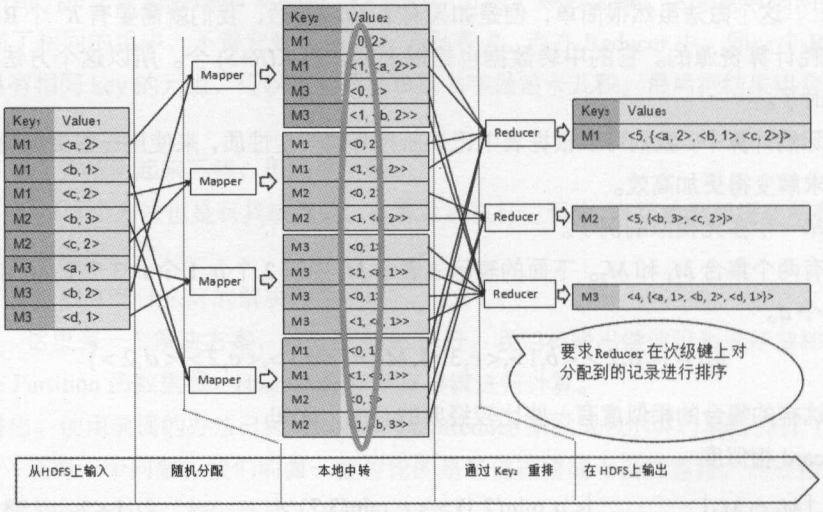
• 析取函数

$$|M_1\cup M_2|=|M_1|+|M_2|-|M_1\cap M_2|$$

有了这些预备知识，我们就可以利用 MapReduce 来增加相似连接的可扩展性了。在这里 MapReduce 实际上完成两个工作，其中一个是计算前面提到的单元函数值，也就是集合的大小；另一个就是计算集合的合取函数值，然后通过单元函数和合取函数求解集合的析取函数值。

小可：那么具体是怎么做的呢？

Mr. 王：我们先来看看求单元函数值是如何在 MapReduce 上实现的吧。



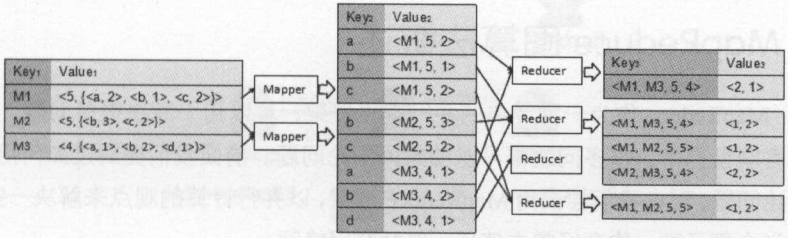
图中有三个集合 M_1、M_2、M_3。键值为集合名称，值为每种元素的个数。在经过 Mapper 的过程中，key 依然不变，但 value 部分我们保存两种 value：第一种用 0 标记，用来统计这个 Mapper 收到的来自某一个集合的元组个数，比如第一个 Mapper 收到了 M_1 两个、M_2 两个；第二种我们用 1 以上的数来做标志位，表示这个 Mapper 收到了多少种来自这一集合的元素，比

如第一个 Mapper 收到了来自 M_1 的一种元素（标志位记为 1），其后面的部分记录各种元素的数量（<*a*,2>），和之前一样。这里包含了一个思想，就是合理设计 value 值的结构，让 value 值可以有多种不同的类型，比如这里设计了一个标志位来区分不同类型的 value 值。同时这个 value 里面有一个次级键的思想，value 里面的第一个值就是一条记录的次级键，在洗牌的过程中，可以实现利用次级键进行二次排序。

接下来数据经过洗牌之后被送到了 Reducer 中，从图中可以看出，Reducer 对数据进行了整理，生成的键值对的第一个 value 属性就是每一个集合的计数，也就是单元函数值。

小可：那求合取函数值又是怎么做的呢？我觉得如果 HDFS 上可以存储前面的输出结果的话，那么求合取函数值时是不是可以对这个结果加以利用呢？

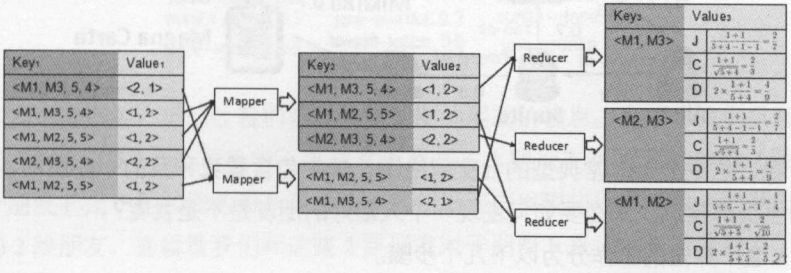
Mr. 王：对。之所以我们在求单元函数值的过程中保留着对各种元素的计数，就是要进一步应用这个结果。



在求合取函数值的过程中，Mapper 做的一件事情叫作交换键值。我们将元素反提出来，将集合名称放进 value 中，变成一种有利于做合取的形态。接下来在 Reducer 中，每一个 Reducer 整理一种元素，比如某一个 Reducer 整理 *a*，这个 Reducer 将其整理成 key=<M_1,M_3,5,4>、value=<2,1> 这种形式。它的意思是，M_1 和 M_3 中分别含有 5 个和 4 个元素，其中 *a* 的个数分别为 2 和 1。

小可：我想做到这一步，合取函数值已经求出来了吧？

Mr. 王：没错，做到这里，合取函数值已经可以通过这一步的结果知道了。我们进一步做下去，再用一轮 MapReduce 将相似度彻底求出来。



Mr. 王：下一轮的这个 Mapper，会把中间结果发送出去，而 Reducer 会收到这些结果，我

们就能求出三种不同相似度计算方法对应的结果。

这里有一点点小思考，我们可以对求单元函数值部分做出改进，让它的求解变得更快速一些。

首先还是要对表进行随机分割，但是在 Mapper 运行之后，我们尝试对 Mapper 生成的数据进行压缩。我们在 Mapper 中只统计元素的个数，而忽略它们具体是什么；在 Reducer 里面做一个求和就可以了。

小可：这样在求单元函数值时的确可以更加节省资源，但是后面的合取求解怎么做呢？

Mr. 王：当我们已经求出了一张表，包含每个集合中的元素个数，这张表保存在每一个 Mapper 中。只要每个 Mapper 中都有这样的一个表，求解相应的占比也就容易得多了。但是对于 Mapper 来说，这样的表也是对内存的一种消耗。好在这个表不会太大，消耗的内存也不会太多。

6.3.2 MapReduce 图算法概述

Mr. 王：MapReduce 作为一种经典的并行编程框架，可以用于解决很多问题，包括一些图论问题。在客观世界中，很多问题都可以抽象为图论问题。前面我们提到过如何用磁盘算法来解决一些图论问题，现在我们尝试用 MapReduce 框架，以并行计算的观点来解决一些图论问题。

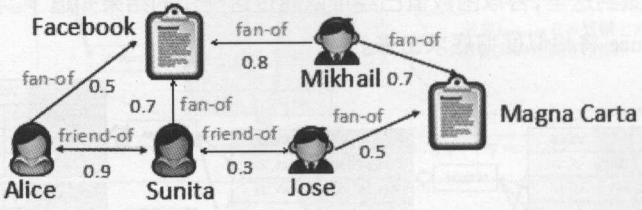
还是先举个例子吧。你会经常去使用一些社交网络吧。

小可：是的，现在通过社交网络，我可以非常方便地与同学联系。社交网络上人与人之间的好友连接关系就可以抽象成一个图。

Mr. 王笑着说：有没有想过，在社交网络上，你的好友的最好朋友是不是你？

小可：哈哈，我没想过，但是这还真是一个挺有趣的问题。用 MapReduce 还可以解决这种问题吗？

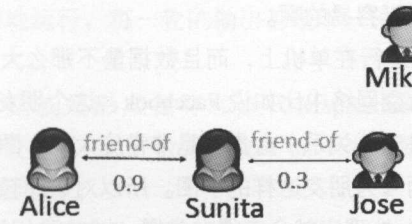
Mr. 王：是的，具体怎么做，我们用一个社交网络图来举例吧。



小可：嗯，这是一个非常典型的社交网络图，其中包含着我和我的一些朋友，还有一些我关注的公用主页等信息。那么要如何发现一个人最好的朋友是不是我呢？

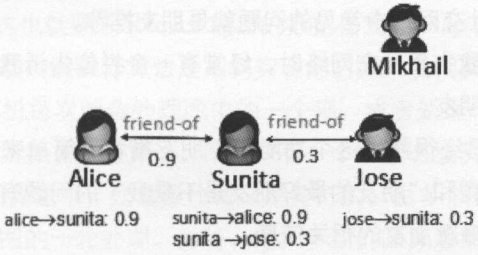
Mr. 王：这个算法的工作分为以下几个步骤。

第 1 步：去掉无关的边和节点。



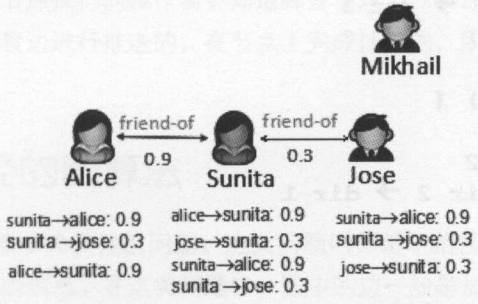
这一步是对整个图进行一个预处理，将图上的一些与我们研究的问题无关的节点去掉，毕竟来自社交网络上的信息还是比较繁多和杂乱无章的。描述社交网络的图上往往会包含一些与“好友”这种关系无关的边和点，会涉及一些关注的网页、兴趣爱好等这样的数据，这与我们要完成的任务无关，为了更好地解决问题，我们要将与好友关系相关的内容具体地提取出来。

第 2 步：用朋友列表标记每一个节点。



对于每一个节点，我们都为其标记一个朋友列表，这个朋友列表记录了与之直接相连的朋友节点，以及自己与他们之间的亲密度。

第 3 步：沿着每条边下推标签。



经过多次下推标签，可以让我们获得关于朋友的朋友的信息。直接与我们具有朋友关系的这些节点是“1 跳朋友”，而与我们不是朋友关系却与 1 跳朋友有关系的就是 2 跳朋友，想要知道某一个朋友 F 是不是和我的关系最紧密，我们就要去发现与朋友 F 是 1 跳朋友的用户，他们是我们的 2 跳朋友，去看看我们和这些 2 跳朋友对于朋友 F 来说的亲密度如何。这样经过比较，我们就可以知道自己究竟是不是他最好的朋友了。

小可：这个算法看起来还是挺容易的啊。

Mr. 王：如果这个算法仅仅运行在单机上，而且数据量不那么大的话，那么这就是一个非常简单的问题。但是在实际的社交网络中比如说 Facebook，这个朋友关系网络就会是一张非常大的图，不论是顶点（人）还是边（关系）数量都是非常庞大的，图的稠密程度也很高，在一个小的圈子内，经常会出现两两互为朋友这样的子图。所以对于比较大的图来说，我们想要执行前面的算法就会遇到很多困难，处理它就会变得非常慢。此时，我们就需要 MapReduce 的帮忙。这个问题用 MapReduce 去做虽然不那么自然，但却是可以解决的。

我们可以将图表示为邻接表，将用户名字作为 key，将其朋友列表和对应的友好度作为 value，形成用户→ < 朋友列表，友好度列表 > 的形式输入到 Mapper 中，Map 执行的操作是，将每一个用户的朋友列表拆开，形成用户→ { 朋友，友好度 } 的形式输出，此时在 Reduce 中会将相同的键值合并，我们只要将他们的 value 进行排序就可以了。

Mr. 王：另外一个在社交网络中常见的问题就是朋友推荐。

小可：这个我知道，我在用社交网络时，经常有一个栏位告诉我有个人与我有很多的共同好友，所以他可能是我的朋友。

Mr. 王：嗯，看来你已经很熟悉这个功能了。朋友推荐，简单来说，就是“谁是我多个好友的好友”问题。这个问题和“朋友的最好朋友是不是我”的问题有一个共性，就是都需要我们去发现我们的两跳甚至多跳朋友的相关信息。

Mr. 王：此时我们用一轮 MapReduce 已经不足以解决多跳的问题，以至于要使用多轮 MapReduce。这种多轮 MapReduce 我们称之为迭代 MapReduce。

下面这是一个迭代 MapReduce 的基本框架。

```
从输入路径拷贝文件 → dir 1
(可选：进行预处理)

while (!结束条件) {
  map 从dir 1
  reduce 到dir 2
  move 文件：从dir 2 → dir 1
}

(可选：后处理)
从dir 2 → 输出路径移动文件
```

小可：哦！我看懂了。首先将整个算法的输入内容放入 dir 1 中，然后会去执行一轮 MapReduce，dir 1 会被输入到 Mapper 中，再输出到 Reducer 中，Reducer 会接收来自 Mapper 的输出作为输入，在处理之后输出为 dir 2。之所以能形成循环，是因为它将 dir 2 的结果又送回了 dir 1，然后程序框架会把 dir 2 再次输入到 MapReduce 中。重复执行上述过程，这样

整个系统就可以一轮一轮地运行，每一轮的输出都是下一轮的输入，也就构成了 MapReduce 的迭代。

Mr. 王：你说的对，这就是形成循环和迭代 MapReduce 的基本思路。另外，为了让 MapReduce 进行得更加高效、顺利，在数据被放入 dir 1 中和最终从 dir 2 中取出来之前，可以对数据分别进行预处理和后处理。

想想在迭代 MapReduce 中需要注意什么问题?

小可：这里有一个循环……所以……

Mr. 王：每一轮循环的输出都会拿去做什么?

小可：每一轮循环的输出都会是下一轮 Map 的输入，所以必须要保证 Reduce 的输出符合 Map 的输入形式。

Mr. 王：很好，这需要对 Map 和 Reduce 的输入输出进行标准化处理，使得每一轮的输出和下一轮的输入相匹配，这也就要求 Reduce 输出的数据格式和 Map 的输入格式是严格一致的。

现在我们来看一看一般的图操作算法是如何实现的。如果我们要执行的这个图操作运行在非并行算法下，那么计算机每次就会处理图中的一个项，或者处理一条边，或者处理一个点。也就是说，只有一个访问游标在执行算法，同时只有一个对象在接受算法的处理或者访问。而在 MapReduce 框架下的图算法中，处理操作往往是并行的，由多个 Mapper 同时处理图的多个部分，以求更快地完成对图的一轮处理。另外，绝大多数的图算法都需要经历多个 MapReduce 阶段，这意味着整个算法的构成可能是多个相同的 MapReduce 过程形成的 MapReduce 迭代，或者是由不同的 MapReduce 过程形成的 MapReduce 链。

在进行 MapReduce 算法设计时，我们需要着眼于两个方面：一是对每一个节点的操作是什么；二是要看对每一个节点执行的操作需要知道哪些信息，以及这些信息在图中距离自己有多远。因为信息往往是沿着边进行推送的，在节点上完成计算的，所以要做到对图中的节点有一个透彻的认识。

6.3.3　基于路径的图算法

Mr. 王：接下来我们看一类具体的问题，这类问题叫作基于路径的图算法。这类算法的目标是计算节点间关于路径的信息。在这类问题中，图中的边一般是加权的，这些权也可以叫作边的标记，包括代价、距离或者相似性等。

小可：边的标记就像社交网络图里面的联系亲密度一样吧。

Mr. 王：是的。这类问题的典型例子就是单源最短路径、最小生成树、Steiner 树、拓扑排序等。这里我们要考虑的核心问题就是，如何将这些算法并行化，以解决对比较大的图的操作算法。

现在我用单源最短路径作为例子来说明如何发现计算过程中的并行化。

解决这个问题的经典算法是 Dijkstra 算法。我们先来看看 Dijkstra 算法在内存中的版本和思想。

Dijkstra 算法是由图灵奖获得者、荷兰人 Dijkstra 提出的，是一个非常经典的求解单源最短路径的算法。它求解的问题是这样定义的：在一个加权有向图 $G=(V,E)$ 中，每一条边都有一个非负实数作为它的权，在图中我们标定一个源点 u，去求解 u 到图中其他所有顶点的最短距离，也就是最短路径的长度。

小可：这个算法还是非常实用的啊，如果将城市之间的交通抽象成一个图的话，那么通过单源最短路径就能求解出一个城市到其他城市的最短距离。

Mr. 王：我们先给出这个算法的伪代码，然后再做解释。它的输入是图 G、起始点 u 和总节点数 m。

```
Dijkstra(G,u,m)
{
    将 u 加入到集合 S 中                                    ①
    对于 G 中的每一个节点 n，将 u 到 n 的最短距离 SP[n] 设为邻接矩阵中 A[u][n] 的值 ②
    循环执行 n-1 次                                        ③
    {
            从图 G 的 V-S 中选出一个 SP[i] 最小的顶点 i，将其加入到 S 中          ④
            对于 V-S 中的每一个顶点 j
            {
                SP[j] = min (SP[j],SP[i]+A[i][j])                    ⑤
            }
    }
}
```

我们来分析一下这个算法。

①处：将初始节点 u 加入到集合 S 中。此处的集合 S 表示的是我们已经访问过的节点，在算法开始时，仅有初始节点被访问过。

②处：这是一个初始化操作，将最短距离先设为源点直接可以到达的顶点的边的长度。至于不可达的那些顶点，则设为邻接矩阵中的值 ∞，它不会对后面的较小值比较造成任何影响。

③处：由于除了 u 之外，还有 n-1 个顶点，所以一共要执行 n-1 次。

④处：开始对还没有被访问过的顶点（V-S 中的那些顶点）进行访问，要选择目前距离源点 u 比较近的那些顶点，因为它们更倾向于帮助发现更近的路径，所以我们是按照距离从小到大的顺序来选择顶点的。

⑤处：当引入了节点 i 之后，我们要看它的引入是不是引起了更短距离路径的发现，所以要比较先经过 I，再由 (i,j) 这条边抵达 j 的路径是不是比当前认为的到 j 的最短距离更短一些，如果是，则说明 i 的引入帮助发现了一条更短的路径，我们就更新到 j 的最短距离；否则，维

持原来的最短路径值即可。

循环结束时，SP[j] 中的值就是源点 u 到 j 的最短距离。

小可：还是挺好理解的，而且设计得非常巧妙啊。

Mr. 王：想想看，这个算法的时间复杂度如何?

小可：假设图中有 n 个顶点，这个算法有两层循环：外层循环需要执行 n-1 次；内层循环的执行是节点数目的线性函数，所以内层循环为 O(n)。综合起来，两层循环就是 O(n^2)。

Mr. 王：不过你只说对了一半。当我们使用邻接矩阵表示一个图时，它的时间复杂度是 O(n^2)；但如果图比较稀疏，边数非常少的话，则还可以尝试用邻接表来表示这个有向图。此时，内循环可以稍作修改，针对边去进行访问，沿着与 i 相关的邻接表进行访问，这样算来，运行的时间与跟 i 相关的边数相关，所有节点的边合成起来就会和图的边数呈线性关系，也就是 O(E)。

小可：哦，原来还有这样的考虑。以后在研究图算法的复杂度时，要多考虑图的稀疏与稠密，还有图的实际存储结构，这些都会对算法的复杂度产生影响。

Mr. 王：内存版本的 Dijkstra 算法每次沿着一个中间节点遍历图，基于总路径的长度去定义遍历的优先级。相当于在这一过程中我们建立一个堆，每次取出堆顶进行处理。在这里面就没有发现并行机制，因为我们每次处理的都是所维护的这个堆的堆顶，每次都在对一个新的顶点进行操作。

为了提升计算效率，我们就要尝试去挖掘其中可能并行的地方。可以想到，从源点出发，能不能不必每次只处理一个顶点，而是每次同时处理一个跳数的节点。比如，第一次处理从源点出发的 1 跳节点，第二次可以从这些 1 跳节点出发，去发现那些距离源点 2 跳的节点，而这些工作之间并不会产生干扰。这样思考的好处在于，我们能够借此发现其中潜在的并行性。并行性在于在下一步开始之前，我们对本轮的这些节点的访问是可以并行进行的。

在传统的算法中，对于 Dijkstra 算法仔细考察每个 u，在其维护的堆中找到堆顶，从而可以安全地删除确定顶点。这部分内容前面已经提到过了，现在要考虑的就是在 MapReduce 中，我们怎么去寻找其中潜在的并行性。

- 对每个 v 考察所有潜在的 u。
- 通过保存 u 的前沿集合迭代计算（距离源点 i 条边）。

小可：那么在 MapReduce 中，具体是怎么做的呢?

Mr. 王：先来想想，要建立一个 MapReduce 解决方案，首先要定义什么?

小可：我想应该是要定义出 key-value 对吧。

Mr. 王：很好。在这个问题中，key-value 对是这样定义的：一个顶点的编号 ID 是它的 key，而 value 包含 < ∞ , -, {<succ-node-ID,edge-cost>}> 这样几个部分：

- 第一个数据域表示从源点到节点 ID 的最短路径长度为∞。
- 第二个数据域表示最短路径上的下一个节点。

小可：嗯，这个时候，最短路径多长还不知道，下一个节点也不知道，这里都初始化成无穷和空。

Mr. 王：succ-node-ID，是其邻居节点的 ID，edge-cost 是到其邻居节点的路径长度。这两部分其实就是节点 ID 的邻接表。

好了，接下来应该做什么？

小可：接下来就要定义 Map 和 Reduce 了。

Mr. 王：嗯，掌握这种解决问题的思维方式，与理解算法是同等重要的。

在 Map 中，每一个 Mapper 接收到的输入都是 ID 作为 key,<dist,next,{<succ-node-ID,edge-cost>}> 作为 value 这样的 key-value 对。但是对于每一个 ID 对应的 succ-node-ID，我们传送 succ-node-ID 作为 key，而 {<node ID,distance+edge-cost>} 作为 value 的 key-value 对。这一步实际上是在做从源点到新发现的这个 succ-node-ID 的路径和路径长度计算。

小可：这时我们是不是应该把最小的代价提取出来呢？

Mr. 王：不，这里并不需要把最短的路径提取出来，但是我们知道，最终要找的最短路径一定包含在这里面。即使提前找出这些最短的路径，也并不一定是最终的最短路径的一部分。

但是仅仅传送这个还是不够的，我们依然需要传送 ID→<distance,nest,{<succ-node-ID,edge-cost>}> 这个 key-value 对。想想看这是为什么？

小可：如果还需要传送这部分，则是因为在下一轮的迭代过程中还需要用到这些节点的原始数据。因为每一轮的迭代都和第一轮所做的计算并无本质区别，在计算下一轮的过程中，所使用的算法和第一轮也是一样的，依然是依赖如同第一轮那样的输入。

Mr. 王：很好。

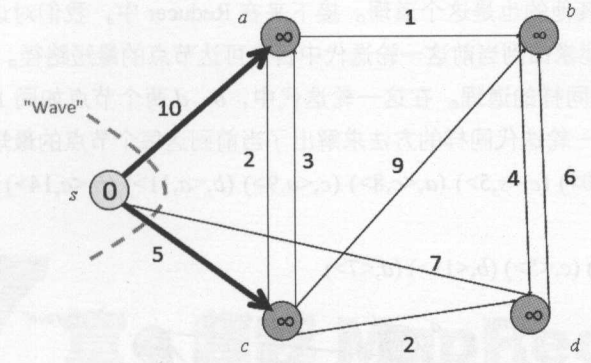
小可：可是这样传送，到了最后节点 ID 和其 succ-node-ID 不是就都混到一起了吗？

Mr. 王：嗯，是这样，因为在 Reduce 中，我们也确实需要这样的机制。

在 Reduce 之前，我们会根据 ID 做一个 Group，也就是将相同的 ID 聚集到一起，然后进入 Reduce，将相同 IDdistance 定义为前驱节点中的最小代价，而 next 定义为具有最小代价的那个前驱节点。最后将这样的 key-value 对传送出去，也就是传送 ID<distance(最小),next(最小),{<succ-node-ID,edge-cost>}> 这种模式。

小可：嗯，似乎是懂了。

Mr. 王：哈哈，“似乎”可不行啊，我们举个例子，把这个问题彻底搞清楚。

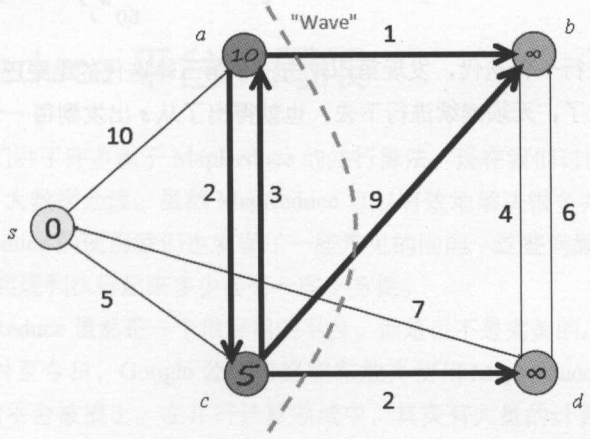


这是一个典型的最短路径问题。

在第一轮迭代中，Mapper 传送出去的数据是 (*a*,<*s*,10>) (*c*,<*s*,5>)，*a* 和 *c* 分别是 *s* 的后继节点，*s* 是源点，5 和 10 是边的权重。

在 Reducer 中，我们求出了到 *a* 和 *c* 的当前最短路径 10 和 5，按照之前我们定义的那种表示法，Reducer 会发送 (*a*,<10, ...>) (*c*,<5, ...>) 这样的数据。

接下来是第二轮迭代。



Mapper：(*a*,<*s*,10>) (*c*,<*s*,5>) (*a*,<*c*,8>) (*c*,<*a*,9>) (*b*,<*a*,11>) (*b*,<*c*,14>) (*d*,<*c*,7>)

Reducer：(*a*,<8, ...>) (*c*,<5, ...>) (*b*,<11, ...>) (*d*,<7, ...>)

在第二轮迭代中，我们重启一次 MapReduce，注意看新增的数据记录，(*a*,<*c*,8>) 这条记录表示，这一次发展到 *a* 节点的邻居是 *c*，而之前到 *c* 的最短路径是 5，*a* 到 *c* 的距离是 3，所以加起来 *s* 到 *a* 的当前最短路径就是 8。

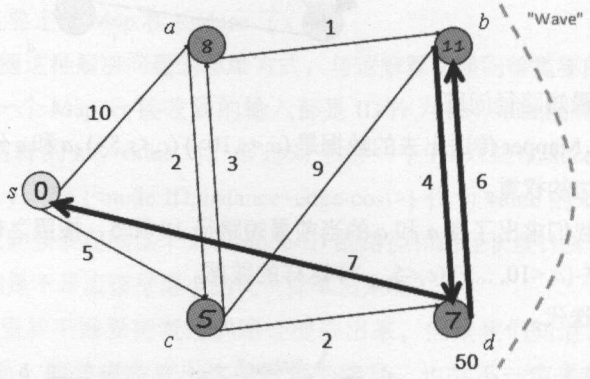
小可：相应的，(*b*,<*a*,11>) 表示发展到 *b* 节点的邻居是 *a*，原来到 *a* 的最短路径是 10，又因为 *a* 到 *b* 的距离是 1，所以当前 *s* 到 *b* 的最短路径就是 11。

Mr. 王：很好，其他的也是这个道理。接下来在 Reducer 中，我们对这些键值对进行基于 key 的分组，这样就能求出到当前这一轮迭代中各个可达节点的最短路径。

第三轮迭代还是同样的道理。在这一轮迭代中，*b*、*d* 两个节点如同上一轮迭代被加入了研究范围，使用和上一轮迭代同样的方法求解出了当前到达每个节点的最短路径长度。

Mapper：(*a*,<*s*,10>) (*c*,<*s*,5>) (*a*,<*c*,8>) (*c*,<*a*,9>) (*b*,<*a*,11>) (*b*,<*c*,14>) (*d*,<*c*,7>) (*b*,<*d*,13>) (*d*,<*b*,15>)

Reducer：(*a*,<8>) (*c*,<5>) (*b*,<11>) (*d*,<7>)



这时我们还要进行一轮迭代，发现第四轮迭代和第三轮迭代的结果已经完全一致，于是认为整个算法已经收敛了，无须继续进行下去，也就得出了从 *s* 出发到每一个节点的最短路径。

第7章 超越MapReduce的并行计算

7.1 MapReduce平台的局限

Mr. 王：前面我们讲了许多基于MapReduce的并行算法，现在我们讨论一个新话题——超越MapReduce的并行大数据处理。虽然MapReduce可以有效地解决很多并行计算的问题，但是经过前面对MapReduce的使用我们也发现了一些常见的问题；这些问题用MapReduce解决虽然是可行的，但是实现和执行起来多少会有一些不方便。

小可：嗯，MapReduce虽然是一个很好用的平台，但是也不是完美的。

Mr. 王：的确，时至今日，Google公司已经宣布放弃使用MapReduce平台了。现在有很多超越MapReduce的平台被提出。在并行计算领域中，其实有大量的计算机工作者在做新计算平台构建的研究，而且目前大多数进行并行计算的科学家和工程师所做的工作其实已经与MapReduce无关了。也有很多好的平台被提出，比如加州大学伯克利分校的Spark等。

小可迫不及待地说：那么继MapReduce之后，哪一个平台是最好的呢?

Mr. 王：这样的平台有很多，它们各有各的适用范围，其实我们并不能说出哪一个平台是最好的，而是在处理具有某种特征的问题时，某个平台的效果就会比较好，因为该平台往往就是面向这种特征设计的。所以并没有万能的平台，合适的才是最好的。这些相比MapReduce来讲比较特殊的平台就像是一种特殊的舞台，某种特殊的舞蹈可能就适合在这种特殊的舞台上表演。

首先我们来介绍基于迭代平台的并行算法。MapReduce 平台已经被证明，作为一个非递归的描述语言通用平台，是非常成功的。后人开发的基于 MapReduce 的各种工具包和组件包非常多，也在并行框架下实现了非常多样和复杂的功能，比如实现类 SQL 语言的 Hive，以及实现嵌套类型支持关系代数的工具 Pig 等。但是 MapReduce 并不是没有缺点的，我们在讨论 MapReduce 中的图算法时发现，在 MapReduce 上运行我们所设计的算法时，往往要进行多轮的迭代 MapReduce，将前一轮迭代的输出结果作为下一轮迭代的输入，不断地执行，这就是循环和迭代。不仅仅是在图的处理中，循环和迭代在程序设计中也是非常普遍存在的，比如在像聚类这样的数据挖掘等中都是非常常见的。而 MapReduce 本身是不能表示循环和迭代的，当需要进行这样的操作时，往往需要在框架之外用脚本来控制。

而循环结果往往是很大的，比如在计算传递闭包、PageRank 这样的算法中，每一轮迭代的输出量都是非常大的，如果平台本身不能提供一个比较好的循环和迭代处理，那么就会非常不方便。另外，每一个循环和迭代算法都要有停止判定，迭代 MapReduce 也不例外，不过在测试迭代 MapReduce 的算法是不是已经收敛时，往往不得不进行一轮额外的 MapReduce，通过观察结果与上一轮是否有区别来判断迭代是否已经收敛。

小可：如果每一轮 MapReduce 操作之后，都要再来一轮 MapReduce 作为退出循环判定的话，那么效率实在是太低了。

Mr. 王：没错，所以我们要考虑一些新的策略和方法。这里我们先提出一个概念，叫作函数的不动点。

小可：我知道，不动点就是能使 $f(x)=x$ 这样的 x 值。可是这和 MapReduce 有什么关系呢？

Mr. 王：你想一想，循环和迭代时，我们一般以什么样的条件作为停止条件呢？

小可恍然大悟，说：当经过迭代之后结果已经不变时，停止迭代。我懂了，x 相当于输入的数据，函数 f 相当于本轮迭代的处理，如果 $f(x)=x$，这说明本轮的输出已经和输入一样，也就是结果不变，这时就可以停止迭代了。

Mr. 王点点头，说：在 MapReduce 这种分布式环境下，我们也要去尝试一些分布式的不动点判定方法，此内容后面再讲。为了设计超越 MapReduce、在循环和迭代的计算中表现更好的算法，我们先来看看现在的 MapReduce 在处理一些典型问题时是怎么做的，在循环上有哪些部分非常浪费资源和效率低下。

先来说说 PageRank。

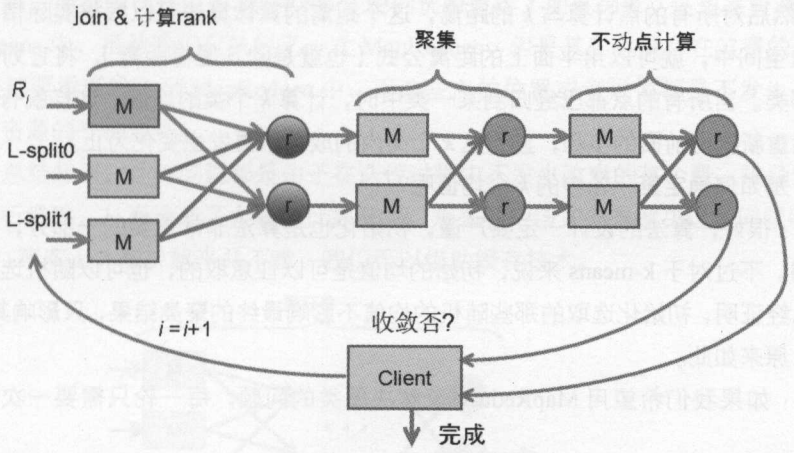
小可：我听说过 PageRank，当年 Google 正是凭借 PageRank 这个简单却非常有效的网页重要度评价算法起家的，一步步成就了今天的 Google。我很想知道它的具体思想是什么。

Mr. 王：PageRank 用于评价网页的重要程度，它的基本思想就是，对于一个网页，如果指向（有超链接连向这个网页）它的网页越多，或者指向它的网页越重要，就说明该网页的重要程度越高。在这个算法的执行过程中，我们就需要不断地根据链接去更新一些网页的重要程度。

在这些网页的重要程度更新之后，它们所指向的网页的重要程度又要由于这些网页的更新而更新，也就需要不断地循环和迭代，一直迭代到这些网页的重要程度不再变化为止。

PageRank 用 MapReduce 来做的话，分为三步：①对存储各个网页及其重要程度的表，与记录链接的表做一个连接，因为有链接，我们就要更新它所指向网页的重要性；②更新每个网页的 rank（重要程度），就根据第 1 步求得的结果来做；③由于要检测收敛情况，所以还要额外地启动一轮 MapReduce 去判定算法是不是已经收敛了。

小可：这样每一轮迭代都要进行三次 MapReduce，开销确实有点大。



Mr. 王：现在我们就要考虑一下，每一轮迭代的开销如此之大，会存在什么样的问题，能不能找到一些可以降低开销的地方。后面我们再说这个问题。

再来看另一个问题，就是求传递闭包。在生活中也有求传递闭包的例子，比如你有 3 个朋友，这 3 个朋友每个都有 2 个朋友，我们就认为，你和这 6 个不直接是朋友的人也是有传递朋友关系的，你们可以通过这种传递关系认识。

小可：这类似于六度空间理论啊。我们虽然不认识，但是却可以通过朋友具有认识这种可能性。

Mr. 王：没错，我们将你的朋友称作“1 跳朋友”，通过你的 1 跳朋友发现的朋友称作“2 跳朋友”，依此类推，可以有“*n* 跳朋友”。当达到第 *n* 跳，这个朋友圈子已经不再扩大时，我们就可以称之为一个传递闭包。这个朋友圈子中的任何一个人，都不能再带来新的朋友，此时我们寻找朋友的这个过程就收敛了。

小可：这个我懂了。

Mr. 王：很好，那你尝试模仿 PageRank，说说在每一轮迭代中，需要哪些 MapReduce 处理？

小可：第一次 MapReduce，要做一次连接，对已经通过传递过程找到的人和他们的朋友列表进行连接；第二次，要做一个去重的工作，因为单纯根据朋友列表进行连接的话，就会发生

重复的情况，比如 A 的朋友有 B 和 C，而 B 同时和 C 也是朋友，那么 C 就会在整个过程中被加入集合两次，这时就需要去重；第三次，进行不动点的验证，我们要去测试朋友的传递闭包是不是已经收敛不再发生变化了。

Mr. 王：很好。其实这个算法和前面的 PageRank 在 MapReduce 执行框架下有很多相似之处，目前设计的这个 MapReduce 也不是很好的，如何去改进，我们一会儿再说。

现在来讨论第三个问题，就是经典的聚类算法 k-means。k-means 希望在一个空间中，这里我们用二维空间举例，将整个空间中的点就近分成 k 类。具体的做法是，每一轮迭代都求出 k 个均值，然后对所有的点计算与 k 的距离，这个距离的具体算法可以根据实际情况而定，在一般的二维空间中，就可以用平面上的距离公式（也就是欧几里得距离），将它划分到距离最近的均值那类。当所有的点都已经归到某一类中时，计算 k 个类的均值，这样就有了新的 k 个均值，然后重新执行前面的步骤，直到这 k 个类内的成员不再发生变化为止。

小可：那如何确定第一轮中的 k 个均值呢？

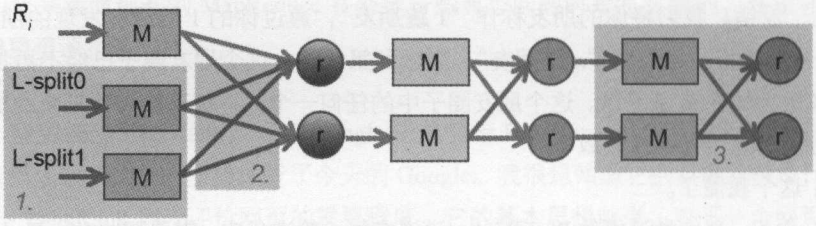
Mr. 王：很好，算法的设计一定要严谨，初始化也是算法非常重要的一部分，在设计算法时不能遗漏。不过对于 k-means 来说，初始的均值是可以任意取的，也可以随机选取数据中的一些点。已经证明，初始化选取的那些随机的均值不影响最终的聚类结果，只影响其收敛速度。

小可：原来如此。

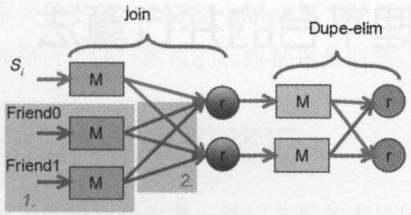
Mr. 王：如果我们希望用 MapReduce 来解决聚类的问题，每一轮只需要一次 MapReduce 即可。

现在我们来看看目前设计的这些 MapReduce 算法的缺陷，也就是它们比较浪费计算效率的部分，分析这些算法不必要的开销，尝试改进提出超越 MapReduce 的算法。首先我们看看比较相似的 PageRank 和传递闭包。

在 PageRank 中我们注意到，记录朋友关系的那张表 L 从始至终是不发生变化的，按照我们之前的设计每一轮都要载入一次 L，这是造成浪费的第一个地方。其二，在每一次迭代的过程中，都会由于 MapReduce 的洗牌而重排 L。另外，在每一次迭代中，不动点计算作为单独的 MapReduce 工作执行，这也是明显的计算资源浪费。



相似地，在传递闭包的计算中也有类似的计算资源浪费情况。

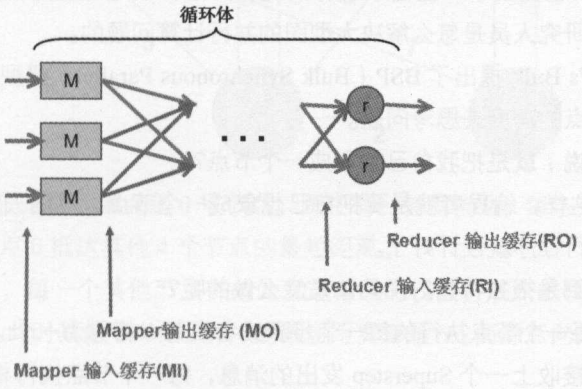


朋友列表在循环中也不会发生改变，而每一次循环都要重新载入朋友，将之输入到 MapReduce 中，而且同样是由于 MapReduce 中的洗牌重排了朋友列表。这也是资源的浪费。

在 k-means 中，虽然我们只执行了一次 MapReduce，但是其实也是存在浪费的。所有点的集合每一次都要重新输入到 MapReduce 中，而这些点的位置或者说值都是不发生改变的，这也是对计算资源的一种浪费。

小可：总结起来，这些问题都是由于在迭代过程中不发生改变的某个量，被频繁重复地输入算法并进行洗牌，从而造成了大量的计算开销。那么我们该如何去解决这些问题呢？

Mr. 王：事实上，解决起来并不难，我们可以借助缓存技术。



我们可以在 Mapper 的输入前加一个输入缓存（MI），在 Mapper 的输出后加一个输出缓存（MO），在 Reducer 的输入之前加一个输入缓存（RI），在 Reducer 的输出后加一个输出缓存（RO）。

小可：这几个缓存如何起到提升算法效率的作用呢？

Mr. 王：我们在执行多轮的迭代 MapReduce 时，相当于进行多轮的循环，而在循环中会有很多并没有发生改变的量，这些量如果每次都重新加载和重排的话，那么对系统的运行效率消耗是巨大的。而缓存恰恰可以有效地解决这些问题。

比如 Reducer 的输入缓存，缓存了那些无须 Map/Shuffle 访问循环不动点的数据。例如在 PageRank 中，输入缓存避免了每一步都重排网络；在传递闭包中，避免了在每一步都进行重排图这样浪费计算资源而没有效果的操作。对于 Reducer 的输出缓存来说，它主要保存前一轮迭代的输出结果，这样系统就可以分布式地访问前一轮迭代的输出结果了。

7.2 基于图处理平台的并行算法

7.2.1 概述

Mr. 王：在实际的应用中，许多计算机问题都会涉及大型图。比如较大区域内的交通网络图、社交网络图，或者是表示一定范围内网站、页面链接关系的网络图等。

小可：嗯，对于社交网络，现在每个人都有几百个好友，一个社交网络图一定很大。

Mr. 王：在这样大的图上进行一些操作或者运行一些算法，就要相对复杂一些。其实前面我们也提到了相关的内容，比如 PageRank、最短路径、连通分量和聚类等。

对于比较大的图，我们很自然地想到可以设计并行算法使其运算得更快一些。但是前面我们也提到了 MapReduce 框架并不适合基于图处理平台的算法。

小可：嗯，因为并行处理需要多次迭代，这导致 MapReduce 的迭代影响到了整体性能。

Mr. 王：我们之前也提出了一些基于缓存的优化方法，但是这些方法仍然不够好。现在我们来看看各大公司和研究人员是怎么解决大型图的并行计算问题的。

1990 年，Valiant's Bulk 提出了 BSP（Bulk Synchronous Parallel）模型这个模型要求并行算法的设计者从图中节点的角度去思考问题。

小可半开玩笑地说：就是把我自己想象成一个节点？

Mr. 王：还真是这样，编程者就是要把自己想象成一个节点，然后去考虑每一轮都需要执行哪些操作，这样并行程序就设计好了。

小可：这个理念倒是很独特啊。那具体是怎么做的呢？

Mr. 王：首先将每一个节点执行的每一个步骤或者说每一轮称为一个“Superstep”，每一个节点在每一轮迭代中接收上一个 Superstep 发出的消息，每一个节点执行相同的用户定义函数，修改它的值或者其输出边的值，并且将消息送到其他节点，这些消息由下一个 Superstep 接收，会在下一个 Superstep 中进行相应的处理，从而改变图的拓扑结构。当没有额外工作要做时结束迭代，也就是所有节点同时变为非活跃状态，而且系统中不再有信息传递时，算法也就相应地终止了。

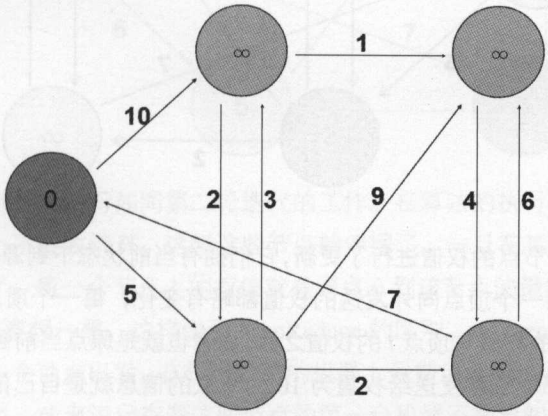
小可：嗯，这样做事实上使得系统达到了更高的分布式程度，每一个节点都不用去关心整个图的情况，只要考虑好自己该做什么就可以了。那么有哪些平台应用了这种思想呢？

Mr. 王：比较著名的就是 Google 的 Pregel，还有 MSRA，也就是微软亚洲研究院的 Trinity 等。这些系统在内部实现上各有侧重，比如 Trinity 是一种基于 Memory Cloud 的系统，它通过网络对各台机器的内存进行汇总综合应用。它的优点就是对于图这种需要随机访问的数据结构，使用内存去存储会让处理效率变得比较高；缺点就是造价比较昂贵，需要硬件设备好一些。

虽然在内部和底层会有很大的不同，但是这些系统提供给用户的接口往往都是一样的，或者说使用它们的方法和在这些平台上进行编程的思想都是类似的。

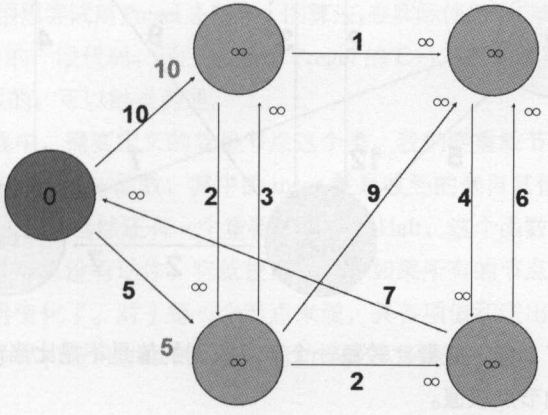
7.2.2　BSP 模型下的单源最短路径

我们先来举个例子吧。单源最短路径也是一种很典型的图论问题，前面我们提到过，就是求解从一个源点到各个节点的最短距离，有时带上求解最短路径。我们来看看如何“把自己想象成一个节点”。以下面这个图为例。

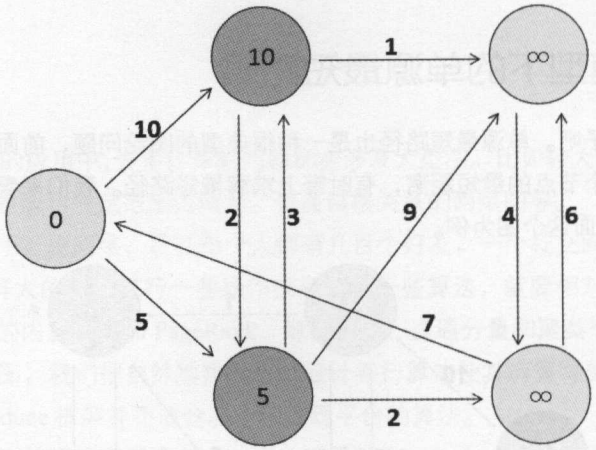


在这个图中，我们选取的源点是 0 号节点。图中有一些有向边，每条有向边都有一个权值。我们要求解的就是源点 0 抵达其他 4 个节点的最短距离。

在第一轮迭代中，每一个其他节点都向外发送自己的权值，节点的权值表示当前状态下源点 0 到它的最短距离。此时其他节点到源点 0 的最短距离都是 ∞。而源点向外发送其出度边的值，就像这样：

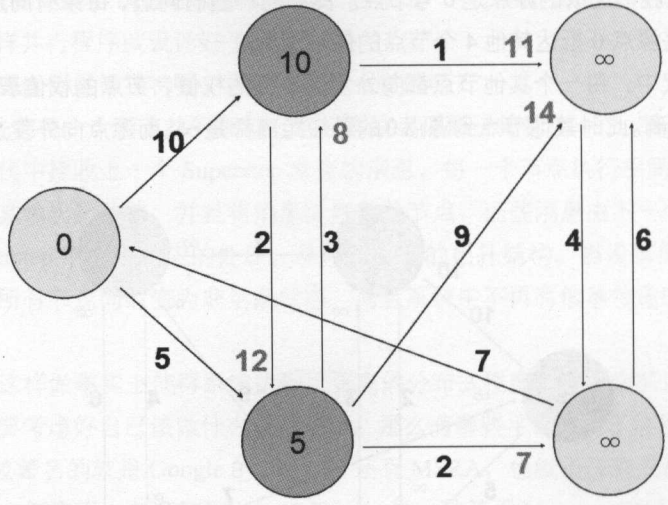


经过了第一轮迭代，图中的权值会变成这样：



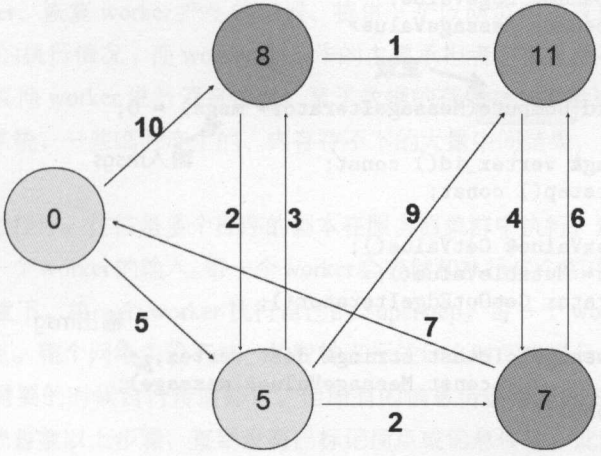
源点 0 右侧的两个节点的权值进行了更新，它们拥有当前状态下到源点的最短距离 10 和 5。

从第二轮开始，每一个顶点向外发送的权值都略有变化，每一个顶点 i 向它的每一个出度顶点 j 发送出度边 (i,j) 的权值与顶点 i 的权值之和，后者也就是原点当前到 i 的最短距离。例如，下图中的权重为 5 的那个节点发送给权重为 10 的节点的信息就是自己的权重 5 与它到权重为 10 的节点的出度边的长度 3 之和，结果为 8。



小可：哦，我懂了，接下来要比较每一个节点收到的值是不是比当前的最短距离要小，如果是，就要替换当前的节点权值。

Mr. 王：没错，经过这一轮替换，结果就是这样：



这样每一轮节点都继续执行如同第二轮迭代的工作。在算法的执行过程中，有些节点在某轮中既不发出消息，也不接收消息，这样这些节点就停摆了。一旦在某轮中所有的节点都停摆了，整个算法就结束了。每一个节点上的权值就是源点 0 到该节点的最短距离。

Mr. 王：现在我们要想一想，这样做和 MapReduce 的区别。在 Pregel 平台上程序设计的最大特点就是从图中每一个顶点出发，在执行计算的机器上保持顶点和边，用网状结构传输信息。也就是说，每一个节点，或者说保存着这些节点的每一台机器不需要知道整个图的结构，只需要执行自己本轮应该执行的操作就可以了。这样做可以说是一种“真正的”分布式思想。

而 MapReduce 则不然，虽然在解决图问题时我们可以设计一轮轮的 MapReduce 去进行图算法中的迭代，但是每一阶段都要利用整个图的全部状态，需要整合 MapReduce 链。这意味着 MapReduce 只是借助了分布式计算平台，让多台机器投入到图算法的求解中来，却不能在出现多轮迭代的图计算中实现“真正的”分布式。

小可兴奋地说：我很想尝试用 Pregel 去写一个图算法，在实际使用中应用 Pregel 是怎么做的呢？

Mr. 王打开电脑中的一段代码，说：这就是 Pregel 的 C++ API。其实 Trinity 平台的 API 和 Pregel 的都是比较类似的，可以触类旁通。

在实际使用的过程中，需要定义的就是节点这个类。我们要重载节点的几个函数，也就是前面提到的执行操作（Compute 函数，其中的 msgs 就是收到的来自其他节点的信息）、发送信息（SendMessageTo 函数），当然还有一个重要的 VoteToHalt，这个函数用来确定整个系统是不是停摆，比如本轮中某节点没有动作，它就投票停机，如果所有的节点都投票停机了，那么就说明整个系统已经不再变化了。对于每一个节点来说，其各项值和送出信息的数据类型都是不同的，在实际的设计过程和使用过程中，可以借助模板类来确定。

```
template <typename VertexValue,
          typename EdgeValue,
          typename MessageValue>
class Vertex {
 public:
  virtual void Compute(MessageIterator* msgs) = 0;

  const string& vertex_id() const;
  int64 superstep() const;

  const VertexValue& GetValue();
  VertexValue* MutableValue();
  OutEdgeIterator GetOutEdgeIterator();

  void SendMessageTo(const string& dest_vertex,
                     const MessageValue& message);
  void VoteToHalt();
};
```

重载

输入msgs

输出msg

小可好奇地说：我还想看看最短路径这个程序具体是怎么写的。

Mr. 王：我这里刚好有一份最短路径的代码。

```
class ShortestPathVertex
    : public Vertex<int, int, int> {
  void Compute(MessageIterator* msgs) {
    int mindist = IsSource(vertex_id()) ? 0 : INF;
    for (; !msgs->Done(); msgs->Next())
      mindist = min(mindist, msgs->Value());
    if (mindist < GetValue()) {
      *MutableValue() = mindist;
      OutEdgeIterator iter = GetOutEdgeIterator();
      for (; !iter.Done(); iter.Next())
        SendMessageTo(iter.Target(),
                      mindist + iter.GetValue());
    }
    VoteToHalt();
  }
};
```

我们一起来剖析一下这段代码。和前面的框架是相符的，继承了图中的节点，派生出最短路径节点类，在派生类中重载了 Compute 函数，程序在执行过程中我们要做的就是遵循前面提到的思想，接收所有节点向图中的某一个节点发送的消息，然后判断这些消息中包含的路径权值是不是比图中的某一个节点小，如果小就更新图中的某一个节点。同时图中的某一个节点也要向其他节点发送包含图中的某一个节点最短路径和出度边长之和的消息。最后，如果本轮没有任何动作和改变，图中的某一个节点要执行投票停机这个操作。

小可：如果我想部署一个 Pregel 平台，需要怎么做呢？

Mr. 王：一般来说，一个 Pregel 平台都需要运行 master 和 worker 两种进程。其中 master 主要完成维护 worker、恢复 worker 产生的错误、提供一个以网页形式表现的用户界面来控制和监督 worker 进程的执行情况；而 worker 是工作的主要承担者，它负责在每一轮迭代中处理自己的任务并且与其他 worker 进行消息传递。至于数据的存储，一般会选用 GFS 和 BigTable 这样的分布式文件系统，一些临时产生的、内存存不下的大量中间结果，会相应地被存储在磁盘中。

在实际的运行过程中，往往是多个程序的副本在服务器集群中执行，然后由 master 进程分配划分内容作为每一个 worker 的输入。每一个 worker 会存储和执行多个图中的顶点并标记它们。在 master 的命令调度下，每一个 worker 执行自己的 Superstep。每一个 worker 循环通过已标记的顶点计算每个顶点。整个网络中没有统一的时钟来标注什么时间去进行信息传递，而是采取信息异步传输，有需要的时候自行传递即可，但所有的信息传递都要在 Superstep 结束前传送完毕。系统会不断地重复以上步骤，直到没有已标记顶点或信息传输，此时我们想要的结果就已经求解出来了。在程序运行停止后，master 会命令每一个 worker 保存图的部分信息。

7.2.3　计算子图同构

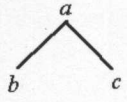
Mr. 王：我们再来看一个例子——计算子图同构。这个问题给定（节点有标签）数据图 G 和查询图 P，找到 G 中和 P 同构的子图。这是一个经典的 NP 完全问题。

小可：那求解岂不是很困难？

Mr. 王：在实际情况下，虽然数据图 G 会比较大，可能有上亿个节点，但查询图 P 一般会比较小，因为查询图一般是由查询需求表现出来的，查询需求往往没有那么大。

小可：如果依然利用 Pregel 平台的思想来解决问题，要怎么做呢？

Mr. 王：考虑到 Pregel 平台具有面向节点编程的思想，我们就要考虑在比较大的图中较小的相邻结构。因为在每一轮迭代处理中，每一个节点也就只能和与其相邻的结构进行通信，所以我们使用一种叫作 STwig 的结构，这种结构就是只有两层的一棵“小”树。

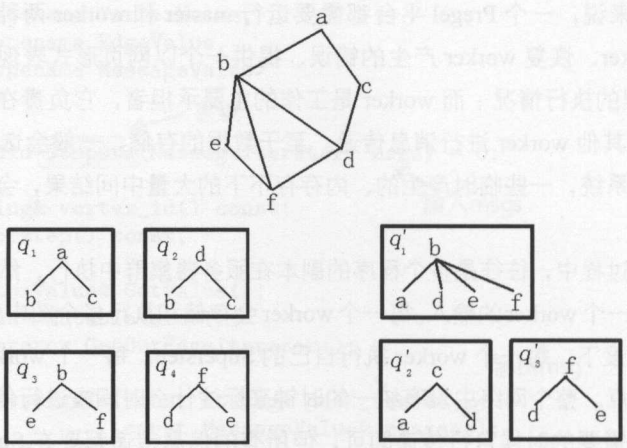


小可：嗯，只有两层的树的确可以很有效地表示某一个节点的所有邻居。

Mr. 王：我们要将整个大图拆分成 STwig 结构，这样做的效率比匹配点和边显然要高得多。然后我们按照 P 中的 STwig 到 G 中去搜索相同的结构。经过数次迭代之后，将查询出的 STwig 再重新 join 成原来要查询的图结构就可以了。总结起来就是：拆分、查询、join。

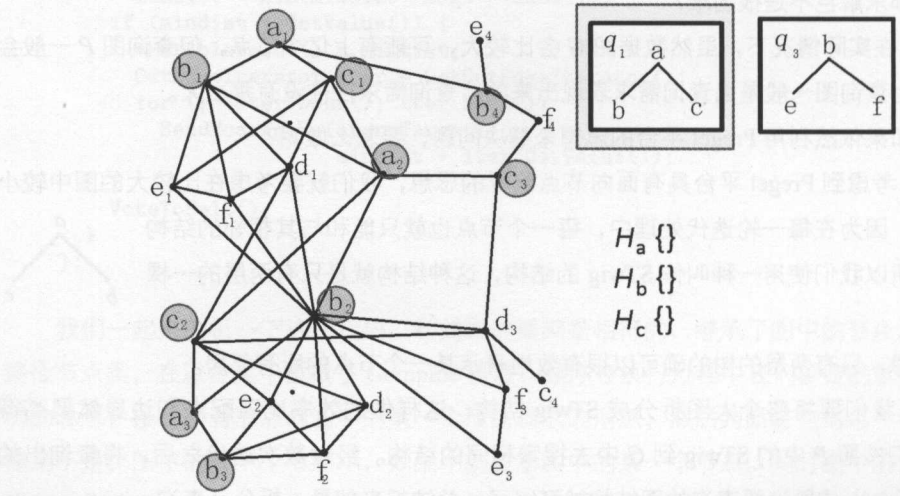
小可：还是有点抽象，具体是怎么做的呢？

Mr. 王：好，我用一种具体的图模式来演示这种思想的具体步骤。



第 1 步：划分。对查询图 P 进行分解。比如上面的图 abcdef 是想要查询的图模式。我们将其分解成多个 STwig，也就是大小只有两层的树。当然，分解方法是不唯一的，而且求解最佳的分解方法也是 NP 完全的，好在图模式的大小比较小，相对来讲比较容易求解。比如图中的 q 和 q' 就是两种不同的划分方法，当我们完成划分之后，这一次查询 Q（或者说查询图 P）就可以表示成 $q_1q_2q_3q_4$ 或者 $q_1'q_2'q_3'$。

第 2 步：搜索。在搜索时，我们先选取一种模式，比如 q_1，然后到数据图 G 中去搜索子模式 q_1。



这个搜索过程可以很好地利用 Pregel 的思想。因为我们要寻找的模式都是两层的小树，所以在搜索 q_1 这种模式时，只需要让每一个节点检查自己是不是 a，然后再让每一个 a 去与其邻居联系，看看它们是不是 b 和 c，如果其邻居中同时有 b 和 c，那么就上报说明自己这里有 q_1。

小可：这样确实可以很好地利用“把自己当作一个节点”的思想。无须关注图的其他部分，也不用从整体上去寻找这种模式，而是每一个节点自己出发去检查其邻居是不是和自己构成了一个 P 中存在的 STwig 结构。

Mr. 王：这里还有一个小技巧。我们不难注意到，在要进行搜索的这些 STwig 中，会存在一些公共的节点，比如 q_1 和 q_3 就有公共节点 b。我们可以做一个缓存，记录下来在搜索 q_1 时找到的所有的 b。当进行 q_3 搜索时，我们无须去搜索所有的 b，而是只搜索 q_1 时找到的那些 b 就可以了。你想想看这是为什么？

小可想了一下，说：能够匹配模式 P 的 G 的子图必然同时匹配 q_1 和 q_3。如果进行 q_1 搜索时并没有将某一个 b 搜索出来，则意味着符合 q_3 的子模式不符合 q_1，因为它的 b 不连着 a 或者它连着的 a 不连着 c。如果是这样，即使它真的能够匹配 q_3，那么到最后组合起来，也会由于不含 q_1 不可能构成 P。

Mr. 王笑着说：非常好，你的逻辑思维很严谨。这样做的好处是，可以大大减少我们在每一轮搜索过程中需要处理的节点。

第 3 步：对查找到的这些子模式进行一个 join 操作，将小的 STwig 合并成更加接近原图 P 的结构，以求能够返回最终结果。这个 join 操作相对会麻烦一些，因为图的连接并不是一个符合“以节点为核心”思想的操作，这里就需要所涉及的每一台机器都要加载一些中间结果，让这些中间结果在自己这里进行连接。

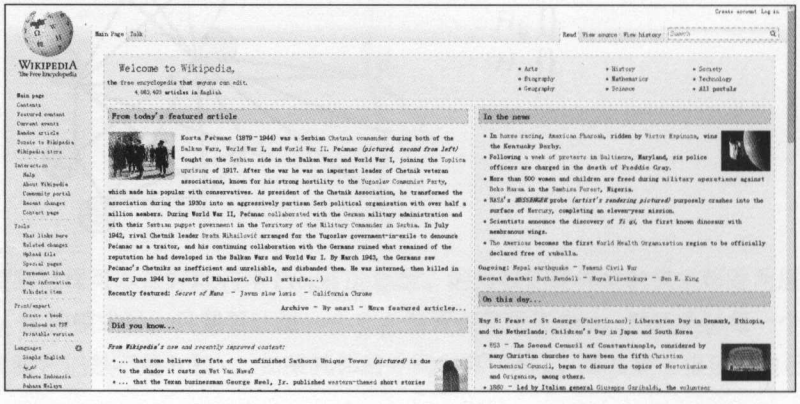
第8章 众人拾柴火焰高——众包算法

8.1 众包概述

8.1.1 众包的定义

Mr. 王：平常遇到不知道的概念或者名词，你一般会怎么办?

小可：有维基百科啊，我去查一查就知道了。对于一个名词，维基百科能给出很多的解释，而且这些解释往往非常准确和专业。



Mr. 王：好，今天我们就来聊聊维基百科。

小可惊讶地说：哦？维基百科还和大数据算法有关？

Mr. 王笑着说：当然有关了，维基百科的策略体现了“众包算法”的思想。

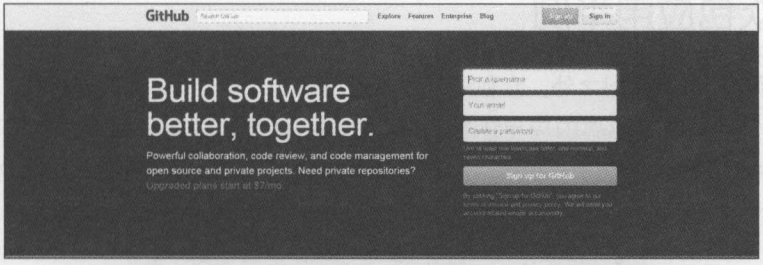
小可：众包算法？什么是众包算法啊？我以前好像听说过有一个说法叫“外包”，就是做一些自己不太容易完成的问题时，会找一些其他的人来做。这个众包和外包有什么联系吗？

Mr. 王：有相似之处，但不太一样，所谓“外包”是把工作交给其他的人来做，但是这些人往往是确定的、已知的这样一群雇员；而众包则不同，虽然它也是把事情交给别人来做，但是这群人往往是不固定的、参与量很大的一群未知的参与者。

小可：原来如此！

Mr. 王：你听说过开源软件吗？

小可：我知道的，一些程序员会在像 GitHub 这样的网站上公开自己软件的源代码，看到代码的人都可以使用和编辑它们。他们拥有自己的社区，会在上面分享自己的一些源代码，同时也就可以基于其他人做的一些前置工作，来更快捷地完成自己的开发工作。有时候程序员还会在开源社区贴出自己的代码，请求别人来帮忙完善。



Mr. 王：其实众包的思想和开源软件很像，只是众包的应用更加广泛一些，它不局限于程序开发，它会将更丰富的、不限于开发的任务交给一些其他的人去完成。就像开源软件一样，究竟哪些人会来完成这些任务，是任务的提出者并不知道的。

小可：可是为什么要采取众包这种方法呢？

Mr. 王：众包算法的问题往往具有这样一个特点，就是人来完成这个问题很容易，但机器做起来却比较困难，或者这个任务难以由少量的人来完成，适合由大量的人参与到其中，充分发挥大众的力量，使得任务更高效、准确地完成，这样的问题非常适合使用众包算法进行解决。

小可：这样就可以发挥人在完成任务上的优势了，这的确是一种非常好的思想。

Mr. 王：想一想，维基百科是不是也利用了众包的思想呢？

小可：的确，维基百科将词条贴到网上，让网络上的人来丰富其解释，是一种众包方法的体现。

Mr. 王：世界上成千上万的人参与建设维基百科的词条，也就建成了一个数据量庞大而且质量相当高的知识库。这些人将自己的知识张贴到维基百科上的同时，也能利用维基百科来解

决一些自己不太清楚的问题。在像维基百科这样的平台中，很多它的贡献者同时也是其受益者。

小可：嗯。发挥网友们的力量，最后在平台上达成一种人人为我、我为人人的效果。

Mr. 王：好，说了这么多关于众包的内容，我们还是尝试给众包下一个定义。众包指的是协调一个群体（一般是互联网上的一大群人）来做微工作（也就是每个人做出一点贡献），完成软件或者单个人难以完成的任务。

小可：嗯，但是协调互联网上的一大群人，还确实有一定难度呢。

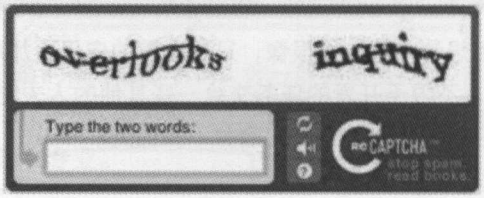
Mr. 王：所以在实际的众包应用中，就要设计一系列的机制和方法来指导和协调群体的行为，如何做到这一点，还值得众包方法的设计者深思。

Mr. 王：这里还有一个概念叫作人本计算，众包和人本计算还是有很大的交集的，但众包和人本计算并不等价。众包在很大程度上利用了人本计算；而人本计算虽然往往是用人来做计算的主体，但人本计算完成的任务可以是大任务，也可以是微任务。众包让一个人做的任务往往是微任务，而合并起来完成的就是大任务。另外，众包算法将任务配发给的人往往是任务的提出者所不知道的，谁来完成这个任务，任务的提出者并不会事先了解或者指定。

8.1.2 众包应用举例

小可：那除了维基百科之外，众包还有哪些应用呢？

Mr. 王：其实众包在业界的应用还是非常广泛的。大量的公司和网站都使用了众包算法，有些众包算法是显性的任务分配和任务处理，也有些众包算法是隐性的。比如这种特殊的验证码：



小可：哦，验证码还是很常用的，只是这个验证码中有两个单词。在登录网站时，为了防止一些自动的脚本攻击网站，会将一个机器难以识别而人容易识别的图像文字放在登录窗口中，只要把相应的文字输进去就可以登录了。

Mr. 王：不错，但是这个验证码比较特殊，之所以使用了两个单词，是因为它有另一个用途。在这两个单词中有一个是真的验证码，用于鉴别正在登录的是不是一个真正的人，这个真正的验证码和其他验证码并无两样；而另一个则是网站希望识别的一个模糊的或者字体比较特殊的单词，比如从古书上、破旧的文章中截取的文本片段。正由于它是比较迷糊的，如果用机器来做文本模式识别的话，就会比较困难，错误率会比较高；但是作为一个人来说，看清楚这个单

词还是非常容易的。网站巧妙地利用了这一点，在用户输入验证码的同时，还帮助网站进行了图像文本识别。

小可：真是一举两得啊，而且其充分发挥了用户的力量，让用户不知不觉间就帮助网站完成了文本识别工作。

Mr. 王：这是一个图像文本识别的例子。在机器翻译中，众包也有很好的应用。时下，机器翻译的质量还没有达到一个非常高的水平，我们常用的翻译平台虽然能够将单词翻译对，但是句式结构往往处理得不够好，语序颠倒的情况时有发生。另外，对于一些意思很多的词汇来说，在特定的语境下识别一个词的意思就很困难了。目前，不仅机器翻译仍然面临着一些困难，而且用机器对人工或者机器翻译好的文章进行翻译质量评价也是很困难的。

小可：嗯，不仅翻译难，而且评价也难。所以，我们就要发挥人的力量，让人进行翻译和评估。

Mr. 王：没错，语言之间的翻译具有一个特点，就是翻译专家和母语的非专家对一个翻译过来的句子的认识是差不多的，毕竟不是专家的人也可以凭借其对母语的了解和熟悉去评价一个句子是不是通顺等。所以不妨将机器翻译或者翻译评价的工作交给那些非专家、网络上的人来做，通过对母语的语感他们就能够很好地评价翻译效果，让人来执行通常会比机器来做好得多。

Mr. 王：其实在图像识别中，众包的例子也有很多。比如我在网上挂一个图，问这是不是“哈工大校园”。

这样的工作让机器来做其实很麻烦，因为哈工大校园里面有各种各样的风景，图像模式非常多，甚至在天气阴晴、季节、相机曝光度和拍摄角度等不同的时候，完全相同的风景都会呈现出不同的效果，让机器来识别难度就非常大。但如果由人来做，就容易多了，不论是什么季节，不论是白天还是夜晚，只要是哈工大的学生或者是去过哈工大的人往往就可以非常容易地辨识出这个图景是不是属于哈工大校园，这种识别的准确度要比机器高很多。

Mr. 王：另外，还有一个很有趣的例子，如果让计算机来识别两张照片是不是同一个人，可能难度就比较大。当今虽然人脸识别技术已经日趋成熟，但是由于光线角度不同、人所处的环境不同，或者是发型、妆容变化等，甚至一张是人年轻时的照片，另一张是中年时的照片，

都会给计算机识别带来很大的麻烦。

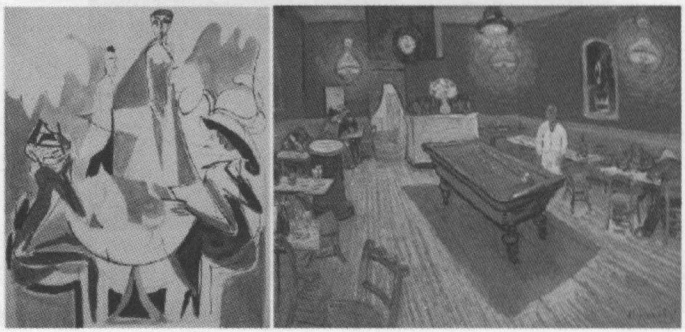
小可：嗯，没错，看看两张照片是不是一个人，对人来说还是挺容易的。即使是年轻和年长时的照片，我们也可以通过观察一些特征，分辨出他们是不是一个人。

Mr. 王：嗯，众包还可以应用在图片分类上。比如这张图片：

小可一眼就认出了图片中的车，说：这不是大众的迈腾嘛！

Mr. 王笑着说：你看，如果让计算机来识别这张图片的话，它可能充其量会告诉你这是一辆汽车的图片，但是如果众包出去，比如交给你，你不仅能看出这是一辆轿车的图片，连它具体是什么品牌、什么型号都可以识别出来，识别的信息精确程度远大于计算机的能力。在实际应用中，众包分类系统可能就会问用户，这是不是一种交通工具啊？用户就会回答“是”；这是不是一辆汽车啊？用户回答“是”；是什么牌子的车啊？……，在这个过程中就像是一棵决策树一样，通过非常简单的步骤，利用人的知识和识别能力，有效地完成了对图像的识别分类任务。

在计算机视觉方面，众包也是有很多应用的。比如给出两张绘画的图片，这两张图片的差异很大，但是我们希望知道这两张图片的绘画风格是不是一样的。



对计算机来说，识别两张图片的颜色是不是一致、描述这两张图片里面的是不是同一件东西或许做起来还可以，但是对于绘画风格这样的概念，是很难让计算机实现的，因为绘画风格这种东西的确很抽象，两张使用不同的色彩、完全不同的绘画，可能同属一个绘画风格，或者出自一位画家之手。

小可：的确，这样的工作由人来做的确可以相对轻松地完成，毕竟人对这种抽象概念的认识还是要强于计算机很多的。如果这个人略懂艺术的话，那么对于绘画风格这样的抽象概念也

就可以在很短的时间内进行很有效的分辨。

Mr. 王：其实在数据库中，也有众包的例子。比如模糊匹配问题。假如有一个人叫张三，在有些数据库记录中，可能被记作“张三”、“Zhang San”、“San Zhang”、“S.Zhang”、“Zhang,San”等，即使两个名字是不同的形式，他们也可能是同一个人。在一些情况下让计算机来做匹配也是有难度的，计算机往往会通过一些形如字符串匹配这样的方法来判断两个名字是不是一个人，这样仍然不够准确。另外，很多名字相同的人，反而不是一个人，不过如果人参与识别的话，判断两条记录是不是一个人，就可以通过比较名字、在现实世界的一定范围内是不是有重名的人，或者是看看各关键字相关的记录，比如头衔、住址这样的信息进行匹配。这些内容有时也是模糊的，比如同一个地址的描述方式有很多，但人依然容易识别它们，可以通过各种模糊逻辑处理方法来解决匹配问题。所以有些系统会借助人的力量来完成这种工作，让人参与到数据库记录的匹配和连接中去，实现更高精度和更好效果的连接。在这方面典型的例子是 CrowdDB，如果你感兴趣，可以查阅关于 CrowdDB 的一些论文。

8.1.3　众包的特点

Mr. 王：我们讨论了这么多众包的例子，现在来研究一下众包的一些特点。你先来说说，一个众包算法需要由哪些部分组成?

小可：首先要有一批请求任务的人；其次要有一群完成任务的人；还应该有一个管理任务的平台。请求任务的人把任务发布到平台上，平台会去搜寻有兴趣来做这些任务的人，然后这些工作者将答案返回给平台。平台收集了大量的答案之后，还要将答案交给提交任务的人。

Mr. 王满意地说：总结得不错，这些提交任务的人，我们一般称之为请求者。而这些完成任务的人，我们称之为工人。整个结构梳理得很清楚，不过，平台的工作还要多一点。首先，平台搜寻工人这项工作就非常需要深入研究——究竟什么样的人适合来完成这样的任务，怎么有效地吸引这些合适的人来完成任务。另外，更重要的是，返回结果的质量控制是平台的关键任务之一，因为众包会将任务交给大量不认识的，来自网络的一群人，这群人中有的真的了解任务的答案并且很负责任，他们给出的答案往往就比较好；反之，有些人并不知道答案，就给出了一些他们认为对的答案或者随便写的一些答案，甚至还会有人故意去破坏平台，给出一些错误的答案，这样得到的答案质量就可想而知了。

小可：确实，将任务交给一些不认识和不确定的人来做，质量控制真的很有必要。

Mr. 王：在众包中，任务的基本单位是 HIT(Human Intelligent Task)，一般也称作“智能任务”。比如一个工人完成了一个任务，我们称之完成了一个 HIT。任务的请求者可以通过 Web 服务的 API 创建一些 HIT 供工人去完成，工人会登录网站，选择自己感兴趣的 HIT 来执行，然后平台和请求者对结果进行评价，给出反馈。据不完全统计，至少有来自 100 个国家的 1 000 000 名以上的工人参与到以百万计的 HIT 中。

Mr. 王：其实工人参与到众包任务中也是有相应回报的。比较直观的回报就是金钱，在有些众包平台上面，工人完成了任务之后会收到一定的资金报酬；而有些众包任务虽然并不能提供报酬，但是很多人依然乐此不疲，是因为这些众包任务设计得比较有趣，很多人为了打发时间，也会去完成众包任务。另外，做众包任务对用户来说也是有一定的社交需求的，同时可以建立自己的声望和好评，获得成就感。

小可：是不是就像百度知道这样的，回答问题赚取分数，就相应地证明了自己的能力?

Mr. 王：百度知道是一个很典型的例子。也有很多众包平台像我们前面提到的双重验证码一样，它并不直接地展现这是一个任务，而是表现为用户平常执行某项工作的一个副产品。当然，众包平台很多的是一种自服务资源，比如最典型的维基百科，也是人人为我、我为人人。大家一同创建这样一个平台，大家也一同使用这样一个平台。

Mr. 王：这里有一个很有意思的问题，假如完成任务的奖励是奖金的话，你觉得是给得越多越好吗?

小可：我觉得应该是吧，重赏之下必有勇夫嘛。

Mr. 王：事实上还真不是这样，在实际的应用中，如果钱给得太少了，会吸引不到足够的参与者；钱给得太多了，就会吸引到很多的滥竽充数的参与者，他们实际上并不能很好地完成工作，只是为了奖金进行尝试，付出过多反而构成了一个反激励效果。准确地说，不是给的钱越多越好，而是钱给得越多，就会吸引到越多的人参与到工作中来。

Mr. 王：在实际的众包任务中，需要考虑的问题还是不少的。比如当我们有任务要完成时，是选择交给现有的众包平台，还是自主开发搭建一个众包平台；而且在众包平台设计时，人机交互是很重要的，应如何激励用户参与到众包任务中；界面的设计是否吸引人、与工人间的交流通信，以及平台的信誉和工人挽留机制都要考虑。质量的检测也很重要，检测垃圾结果、平台可靠性等问题也要考虑。而且如果一个平台上搭载了多个任务，如何对这些任务进行管理，以及人和机器间的平衡，究竟什么任务要人来做、什么任务让机器来做都是众包设计者需要设计的问题。

Mr. 王：下面来说说上面谈到的几个话题。首先是人机交互，众包算法的人机交互非常重要，如果用户体验不够好，对众包任务的参与度和完成质量的影响是非常大的。这方面的解决办法和软件工程中的一些方法是相似的。比如调查、先做出一个原型系统，让一些测试用户进行试用，或者所谓的可用性测试、认知走查等。

Mr. 王：不仅是在完成任务方面，提供任务方面的人机交互也是要非常注意的。平常用搜索引擎时，后面的内容你会关注吗?

小可：对于一些匹配比较好的结果，我一般只看搜索引擎返回结果的第一页，即使是匹配效果不太好的，我也只看到第二页。

Mr. 王：众包平台上的任务也是如此，如果一个众包平台上堆积了大量的任务，统计发现

第三页以后的 HIT，基本不会有人来做，许多这样的任务放在那里一个月都不会有人来完成。设计得不好的提供任务界面，对工人和请求者来说都是非常不利的，任务得不到完成，工人找不到任务。一个任务提供平台一定要有效地将工人和任务进行很好的匹配。

小可：那么网站一般是怎么解决这个问题的呢?

Mr. 王：其实解决这个问题的方法非常多，你也发现了，这种任务交互平台和搜索引擎，甚至如淘宝网这种购物网站都是有相似之处的，很多用于搜索引擎和网购站点的推荐机制都可以用来解决众包任务交互问题。

小可：那么对于质量控制方面的问题，平台一般是怎么解决的呢?

Mr. 王：显然，工人完成任务的质量是众包极其重要的组成部分。但是质量控制并不仅仅是工人的责任，和购物网站一样，不好的卖家和买家都是存在的。众包平台也是一个双向评价的平台，请求者可能会认为工人做得不好，而工人也可能会认为任务是由一个糟糕的请求者发布的。

小可：那平台常用的质量控制方法有哪些呢?

Mr. 王：最典型的方法就是支持率。比如我提出一个问题，8 个人选 A，2 个人选 B，平台会倾向于认为 A 是正确答案。当然，这也不是绝对的，比如统计发现，某些特定的工人经常存在于少数答案的群体中，这些人有可能是垃圾发送者或者破坏平台的人，也有可能他们给出的答案才是对的，请求者提出的问题确实比较困难，大多数人给出的答案都不够准确，而只有这些人给出的答案才是标准答案。不论是哪种情况，这样的群体都是一个特殊的群体，即使表决法不能确定这些人到底是哪类人，但至少可以通过统计发现这些特殊的答案，请求者可以介入，根据统计结果进行分析，进行更好的质量控制。

小可：嗯，单纯地凭借给出某一种答案的数量确实不能确定结果是不是真的准确，真理有时还掌握在少数人手中呢。

Mr. 王：有时众包平台也采用一种准入机制。比如想参与到众包任务的贡献中，先要经过一个资格考试来验证该工人是不是具有完成任务的资格。

小可：这个好麻烦啊，在参与任务之前还要参加考试，这会在很大程度上降低众包任务的完成效率。

Mr. 王：没错，资格考试的确具有这样的缺点，会影响到工人参与的热情，也会耽误任务完成的时间。最重要的是，对于一些主观性任务，或者任务的主题比较分散，设计问题也会很麻烦，需要很大的成本。资格考试还是有很多优点的，很多时候资格考试使用和完成任务一样的方法，这里一方面是测试用户的资格；另一方面也是在教用户如何使用这个平台，用什么样的方法来完成任务，工人在开始任务之前可以很有效地通过资格考试来熟悉任务的完成流程。在解决工人兴趣的问题上，可以调整及格标准或者题量，当工人参与度不高时，可以适当让通过考试容易一点。

小可：如果资格考试或者质量测试发现了糟糕的工人，一般怎么办呢?

Mr. 王：这的确是个问题，对于一些做了不达标工作的工人，应不应该给予相应的奖励，很多众包设计者也有过激烈的讨论，有些人认为，用户提供的坏答案被采纳了，说明系统有问题；也有些人认为，如果对这些不好的工人都给予相应的奖励，岂不是在鼓励欺诈。这里其实有一个折中的方案，我们先用薪金来举个例子，在具体的情况下可能是分数或者其他的回报。我们可以设立 1 个单位的工资和 1 个单位的奖金，只要工人完成了任务，就支付 1 个单位的工资；而对于那些做得好的工人，再额外支付 1 个单位的奖金，这样做要比完全拒绝给工资和全给 2 个单位的工资效果好得多。

小可：嗯，因为很多工人并不是在故意破坏系统，而是确实对这个问题的理解不太好，很努力地完成了任务结果却不好，这样的工人还是要给一点点回报，别打消了他们的积极性。

Mr. 王：不过，对于那些真的在破坏系统的工人，一定要阻止其继续参与任务。

Mr. 王：至于系统的任务分配，也可以分为推方法和拉方法。在推方法中，系统进行任务的推送，完全由系统决定应该将任务发送给谁；而在拉方法中，系统只提供一个平台，工人自己到平台上去寻找任务。在实际的过程中，也要有推荐机制。平台要通过一些算法去衡量成本、衡量用户兴趣、衡量合适程度等，将合适的任务交给合适的人，又好又快地完成任务。

8.2 众包算法例析

小可：讨论了这么多，我还是想通过一个具体的众包例子来了解一下众包算法。

Mr. 王：好，我们就从计算机的角度用具体的例子来分析一下众包算法。通过我们前面讨论的内容，你能不能想到设计众包算法需要考虑的一些基本问题？

小可若有所思，说：嗯……既然很多众包平台是要支付劳动报酬的，那么最起码的众包算法应该要尽量的省钱吧？

Mr. 王笑着说：没错，这是一方面。另外，众包也要能很好地吸引工人有兴趣参与到任务中。下面的例子可以归结为一个实体识别。你先来看看这张表。

ID	Product Name	Price
r_1	iPad Two 16GB WiFi White	$490
r_2	iPad 2nd generation 16GB WiFi White	$469
r_3	iPhone 4th generation White 16GB	$545
r_4	Apple iPhone 4 16GB White	$520
r_5	Apple iPhone 3rd generation Black 16GB	$375
r_6	iPhone 4 32GB White	$599
r_7	Apple iPad2 16GB WiFi White	$499
r_8	Apple iPod shuffle 2GB Blue	$49
r_9	Apple iPod shuffle USB Cable	$19

小可：购物网站上的商品名和价格表。

Mr. 王：发现了什么特点吗?

小可：比如 r_1 和 r_2 这两个商品，虽然它们的商品名是不同的，但是从分析来看，它们应该是同一个东西，都是 16GB 白色版的 iPad 2。

Mr. 王：不错，这种情况在 C2C 购物网站上非常普遍，很多相同商品的说法不同或者一些参数的顺序不同，甚至很多卖家的商品名里面还包含着广告词。

小可笑了，说：哈哈，两个同样的商品，一个商品名是 ××× 如假包换，一个商品名是 ××× 包邮这样吧?

Mr. 王：是的，其实我们希望完成的任务就是能进行一个聚集或者说聚类，将对应的现实世界中的相同实体的商品放在一类中。

在实际完成个任务时，最基本的有两个选择：由机器来做，或者由人来做。由机器来做主要的方法就是基于相似度的。

小可：就是比较两个商品名有多么相似吧? 把两个商品名看作是一些词的集合，看看这两个集合有多么相似。

Mr. 王：没错，一般来说会用一些相似度函数，比如 Jaccard，它就是用两个集合的交集和并集的比来衡量两个集合的相似度的。然后设定一个阈值，高于这个阈值的，我们就认为它们是同一个商品。至于阈值的确定，可以通过一些机器学习或者数据挖掘的方法，求出合适的阈值。

Jaccard(r1,r2)=0.9≥0.8 √
Jaccard(r4,r8)=0.1<0.8 ×

Mr. 王：如果由人来做，最基本的想法就是，设立一些题目，这些题目给出两个商品名，让用户回答这两个商品名是不是同一个东西。你来分析一下，这样的方法复杂度如何?

小可：如果有 n 个商品名，逐一进行匹配的话，就需要 $O(n^2)$ 级别的计算量。

Mr. 王：如果一个 HIT 给 0.01 元，有 10000 个商品呢?

小可：$10000^2 \times 0.01=1000000$ 元。这个开销可太大了，做这么简单的工作要花上这么多钱，太不值得了。

Mr. 王：所以我们尝试改进它，提出一种基于簇的 HIT。我们并不将商品两两进行匹配，而是将几个商品名放在一起，让用户选择其中哪几个是一类的。

如果每次给用户 k 个商品名，那么一次就可以完成 k^2 个对的识别，也就是说，任务量的复杂度可以下降到 $O(n^2/k^2)$。假如有 10000 个商品名，每一次给用户 20 个商品名呢?

小可：$n^2/k^2 = 10000^2 / 20^2$，再乘以 0.01 元，这样花费就小多了，只需花费 2500 元就可以解决这个问题。

Mr. 王：从这个问题可以看出，人和机器是各有利弊的，机器的特点在于，节约金钱成本，而且处理时间比较快；而人的特点在于标注的质量比较高。所以众包算法期待的就是能结合机

器和人的优点，使得成本、时间和质量都达到一个比较好的结果。

小可：那么具体要怎么结合呢?

Mr. 王：其实众包算法中包含的思想就是混合人和机器的工作流程。首先用机器去做一个预处理，可以用前面的某个相似度函数，如 Jaccard；但这次我们不去寻找那些相似度较高的商品名，而是排除那些相似度太低，也就是小于一定的阈值，我们认为已经不可能是同一类的这样的商品名。然后将剩下的部分交给人去进行识别。相应的，也可以使用前面提到的基于对的 HIT 和基于簇的 HIT。基于对的 HIT 可能没有太多的问题，但是基于簇的 HIT 还是有很多方面的东西需要考虑。

小可：比如簇的大小?

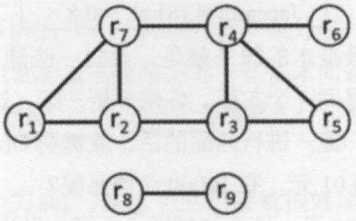
Mr. 王：没错，因为一次用户可以接受的任务中包含商品名的数量是有限的，比如一次给用户 100 个商品名，那用户看起来眼都花了，就会造成匹配的质量大幅下降。也就是说，簇的大小存在一个阈值。另外，每一簇的东西要尽可能地“像一点”。

小可：否则，有可能一簇有 10 个商品名，结果分出 10 组来，它们当中没有一样的东西，这样人工识别就失去意义了。

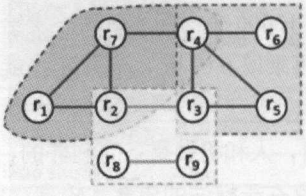
Mr. 王：还有一个问题，就是对于任意两个商品名，它们至少应该同时出现在一个簇里一次。也就是说，要给这两个商品名被识别出是一个商品的机会；否则，结果中就丢失了一条记录。

小可：想不到就连众包里面都有这样的优化问题。那么它怎么解决呢?

Mr. 王：在一些文献中给出了这种问题的解法，就是将这些商品名抽象成一个图。



比如 r_1 和 r_2 的相似度大于某个阈值，我们认为它们存在成为同一个商品的可能，就在图上画一条线；其余的依此类推。这样的话，连通分量有两种：一种是小连通分量，比如 r_8 和 r_9；另一种是较大的连通分量，比如图中其余的部分。此时这个问题就变成了图的划分问题。比如在这个图中，我们就可以将其划分为以下三个部分。

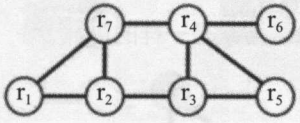


想一想，根据我们前面形式化的内容，三个簇应该满足什么条件？

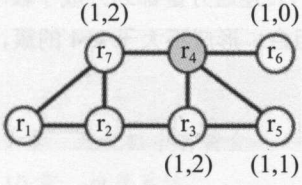
小可：首先，应该满足每个簇的大小不大于簇的大小阈值 k；其次，所有的簇都应该包含图中的每一条边。这样就能保证给每两个商品名被识别出是一个商品的机会。

Mr. 王：不难发现这样的划分方法是一个 NP-hard 问题。要解决它，提出这个问题的论文作者给出了名为“双层法”的解决方法。比如对于上面的图，设 k=4。对于较小的连通分量，不需要考虑，因为它可以被放进一个簇中；对于较大的连通分量，是必然要对其进行划分的。

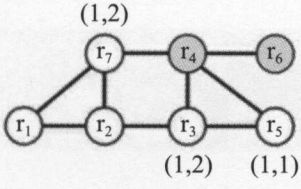
这里我们使用一个贪心算法来对大簇进行划分。使用下图作为例子。



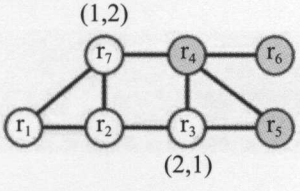
首先，我们找到一个度最大的点，此图中就是 r_4。然后，对和 r_4 相连的点进行标注，标注包括两部分：和簇内点相连的边数、和簇外点相连的边数。



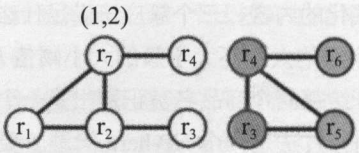
这个序对从意义上反映了它和簇的关联程度。接下来选择和簇内关联度高的点。簇内关联度是一样的，那么选择和簇外关联度低的点，相对来讲，这样的点和这个簇的连接比和其他的簇的连接更紧密，比如这里的 r_6。



接下来用同样的方法，选择 r_5。这里要注意，选择 r_6 对其他点的标注对没有影响，而 r_5 则不同，需要与之相连的 r_3 和簇内点相连的边数进行更新。

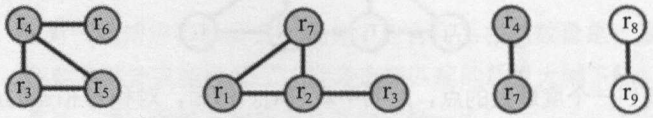


于是我们选择 r_3。此时第一个簇就生成了。



但是这里要注意，虽然 r_3,r_4,r_5,r_6 已经生成了一个簇，但是边 $<r_7,r_4>$,$<r_2,r_3>$ 仍未被包含进这个簇中，这样这个簇就没有给 r_7 和 r_4 商品名是同一个商品的机会，所以我们仍要在图中的剩下部分保留 $<r_7,r_4>$,$<r_2,r_3>$ 这两条边。

剩下的 5 个点，可以继续划分，方法是一样的。



至此，我们就成功地将所有的大连通分量都划分成了较小的连通分量。剩下的工作就是尽可能地将这些小的连通分量进行组合，形成不大于 k=4 的簇，问题也就相应地解决了。

第3篇　应用篇

第9章 大数据中有黄金——数据挖掘

9.1 数据挖掘概述

Mr. 王：今天我们来讨论一个新的话题，你听说过数据挖掘吗？

小可：这个名字倒是挺有意思的啊，不过数据是一种抽象的、虚拟的概念，要怎么去挖掘呢？

Mr. 王：数据挖掘是时下非常热门的一个领域。在大数据时代的背景下，数据量变得非常大，不过我们现在处于一种拥有的数据量大而“知识”匮乏的状态。

小可：这个“数据”和“知识”分别怎么解释呢？

Mr. 王：比如某商家存有大量会员的信息数据，现在公司有一种新产品，他们想知道这些会员中哪些人有更大的可能性去购买这种新产品，从而有效地制定下一步营销战略。从直观上看，仅仅通过这些数据是很难看出来的。对于比较大的企业，它的会员数量也会非常大，用人工去分析海量的数据是不现实的。

再比如银行存有大量的会员用户信息，如工作、信用记录、年收入等，他们希望通过一种手段来鉴别贷款给哪些用户风险会比较小，而哪些用户是高风险的，贷款给他们容易形成不良贷款。这是银行必须考虑的问题，有助于保护银行的信贷资金安全，降低银行运营风险等。但是大银行的会员用户是非常多的，处理数据难度是非常大的。

还比如有一家游戏公司，希望发现每天游戏玩家大多集中在什么时候登录、登录的时间有多长。这样的数据有助于该公司去平衡游戏服务器的负载，对于调整维护和客服人员的工作时

间也是很有意义的。但是比较火热的游戏的登录用户非常多，而且经常玩的玩家登录的次数也会比较多，这就形成了海量的用户登录数据，人工去分析这些数据是很困难的。

在这三个例子中，他的手中都拥有数量规模很大的数据，可是虽然拥有这种大数据，却不能直接对这些数据加以利用。另外，他并不关心这些数据中的每一个细节，他们需要的是一种“知识”，这种知识是来源于大数据的有价值的信息，只有这些有价值的信息才能有效地帮助他们去进一步指导策略或者行动。而数据挖掘，就是从数据中去发现有价值的信息这样一种研究领域。

小可：听起来就觉得这是一项很有意义、有价值的工作啊！不过，数据挖掘要怎么进行呢？或者说这样的“知识”应该如何去发现呢？

Mr. 王：我先举个最直观的例子。比如下面是一组来自某商场销售记录的片段。

年龄	36	35	32	33	21	20	25	23	26
购买产品 A	否	否	否	否	是	是	是	否	是

假如你是销售经理的话，你会更倾向于在哪些人中销售产品 A 呢？

小可：那当然是 20 多岁的人了。从前面的销售记录来看，30 岁以上的人基本上不会购买产品 A，而 30 岁以下的人购买产品 A 的很多。

Mr. 王：其实数据挖掘期待的就是这样一种知识的发现。通过对这组数据的分析，我们发现是否购买产品 A 与年龄呈现一种高度相关的分布。发现 30 岁以下的人更倾向于购买产品 A，而 30 岁以上的人不会购买产品 A，所以我认为，在其他的人群和以后的销售中也会呈现这样的分布。因此，我要更集中地在 20 多岁的人中推销产品 A。这就是我挖掘这组数据得出的结论。

当数据量变得不是 10 条左右的数据，而是达到 PB 级的记录时，单单凭借这些杂乱无章的数据，我们很难利用它们。所以我们需要一种手段，将大批量的数据有效地转化成人类能懂、有实际价值意义的这样的知识，以指导我们进行以后的一些行动。这种手段或者说技术，就是“数据挖掘”。

9.2　数据挖掘的分类

小可：哦，这样就清楚多了。那么在计算机中，数据挖掘的具体方法都有哪些呢？

Mr. 王：一般意义上，我们将数据挖掘算法分为两种，一种是聚类算法，一种是分类算法。聚类算法是对我们要分析的数据直接进行类别划分的那些数据挖掘算法。聚类算法的代表有 k-means、k- 中心点、DBSCAN 等。而分类算法是，首先用一组样本对机器进行一个“训练”。也就是说，首先从这组样本中发掘出分类的标准，然后再对我们要分析的目标数据进行类别划分，从而得出结论。分类算法典型的代表是决策树构建算法、Naive Bayes（朴素贝叶斯分类器）等。

分类算法使用的样本称作“训练集”。出于有无训练集的区别，我们也称聚类算法是无监督学习，因为它没有训练集对其进行一个训练，直接在目标数据上进行操作；分类算法我们称为有监督学习，它有一个训练集对其进行训练，可以先得出一个从样本中提取出的模式，然后再对目标数据进行分类。

比如在前面的例子中，我们可以把这个销售记录表当作训练集，交给数据挖掘算法，经过数据挖掘算法的分析和处理，我们就可以得到一个知识，这个知识就是大于 30 岁的客户大多不会购买产品 A，而小于 30 岁的客户倾向于购买产品 A。在处理以后的数据时，就可以用这个标准来判断。比如来了一个 25 岁的会员顾客，我们就可以把产品 A 推荐给他。

小可：因为有训练集，所以它是一个分类算法。它做的事情就是分析样本，得出知识，然后对目标数据进行一个分类吧？

Mr. 王：没错，分类算法就是在训练集的监督下，得出对后来再出现的数据的一种分类标准。

小可：那么数据挖掘这个过程具体是怎么实现的呢？就是数据挖掘算法的内部具体是怎么做的呢？

Mr. 王：下面就介绍几种经典的数据挖掘算法。另外，数据挖掘操作的集合往往是比较大的，所以我们依然要想到，如何用大数据的思想去处理数据挖掘问题。在介绍完每个数据挖掘算法之后，我们就谈谈如何用适用于大数据的并行平台 MapReduce 来进行数据挖掘的方法和策略。

9.3 聚类算法——k-means

首先我们从聚类算法说起。前面讲过，聚类算法是在没有训练集的情况下对要分析的数据进行一个类别划分。简单来说，就是直接观察数据的分布，将它们“聚集”成多个类别。聚类算法最经典的一个问题叫作 k-cluster。简单来说，就是现在有一批数据，我们要根据这批数据的值将它们划分成 k 类。

对其进行一个形式化的定义，就是：

输入——在一个 n 维特征空间里面的数据项集合。

输出——划分为 k 个类别的数据项。

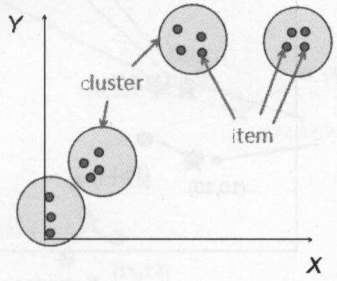
小可：这个 n 维特征空间是什么？

Mr. 王：有一个数据域的数据我们叫它一维数据，有两个数据域的数据就是二维数据。相应的，n 维数据有 n 个数据域，n 维数据可以取值的范围就是一个 n 维空间。比如最简单的二维数据就是平面上的点，它有横坐标 x 和纵坐标 y，它们构成的数据对 (x,y) 就在一个平面上，这个平面就是一个二维空间。在比较复杂的数据中，可以是温度、质量、体积、硬度这样的高维数据，它们就在温度、质量、体积、硬度取值范围组成的一个四维空间里面。我们要做的，就是对这个空间内的这些数据进行一个类别划分，分成 k 类。k 的值是预先确定的，比如现在

平面上有 100 个点，我们要将它们分成 3 类。

小可：如果是把平面上的点分成 3 类的话，我就看哪些点离得近，在图上看是一堆，那就是一类呗。

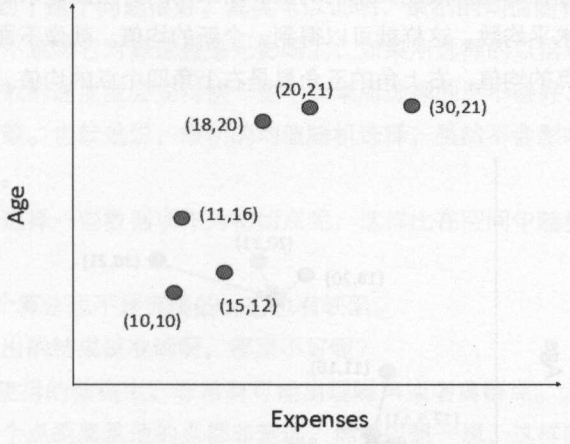
Mr. 王：其实这就是聚类算法的思想。我们直观感受到的那些距离近的点，在各个数据维度上就会比较相近，我们就更倾向于认为它们是一类的。



这就是一个非常典型的 k-cluster 问题。在一个二维空间 *xOy* 中，有很多个点，这些点就代表有 *X* 和 *Y* 这两个数据域的一些数据项（item），而它们就可以直观地根据距离进行一个聚类划分，变成 cluster。

小可：那么能将数据项划分成 cluster 的具体算法是什么呢？

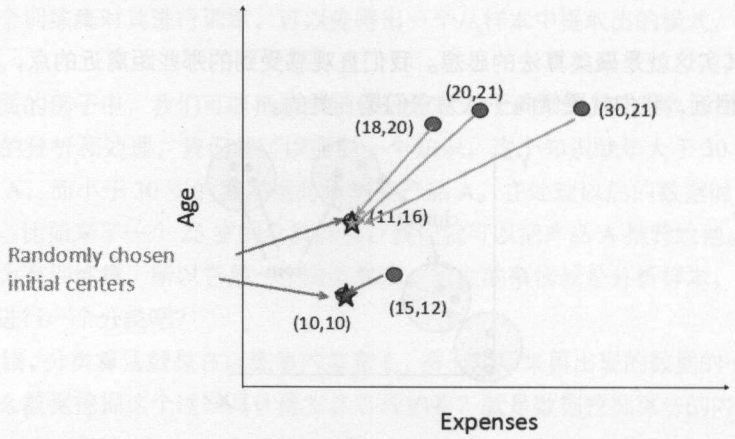
Mr. 王：在聚类算法中，最经典也是最具代表性的就是 k-means 算法，也称作 k- 均值算法。为了方便起见，我们用二维空间进行举例，通过一个实例来看看 k-means 算法是怎么做的。假设有这样一组数据：



这是由年龄和支出组成的二维空间，空间中的点都是由（年龄，支出）这样的二元组构成的数据项。如果 *k* 为 2，也就是将这些点分成两类，我们看看 k-means 是如何解决的。

第 1 步：在空间中随机选取 *k* 个点，在这里由于 *k*=2，所以我们选择两个点。一般为了方

便起见，所选择的点就是某一个数据项，所以这一步也可以改为从数据项中随机选取 k 个点。所选出来的 k 个点，我们称之为均值。

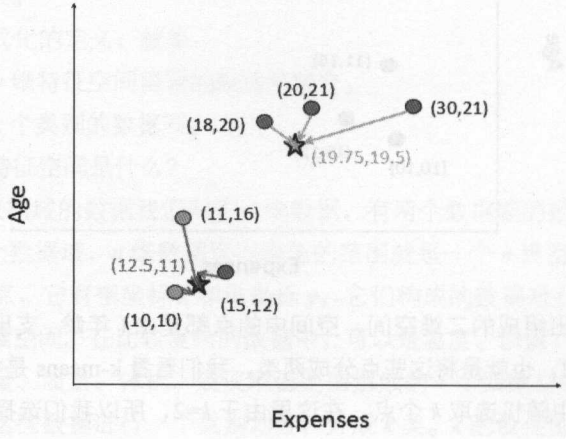


第 2 步：现在有了 k 个均值，接下来让所有的点去计算其与哪个均值的距离更近。每一个点都会选择距离最近的一个均值，将其归入到该均值代表的类别中。

小可：从目前的结果来看，左下角的两个点 (10,10),(15,12) 被归为一类，(11,16),(18,20),(20,21),(30,21) 这四个点被归为一类。但是好像不是很准确，(11,16) 这个点还是归入到左下角这一类更加准确，左下角的这三个点从直观上来看更加接近啊。

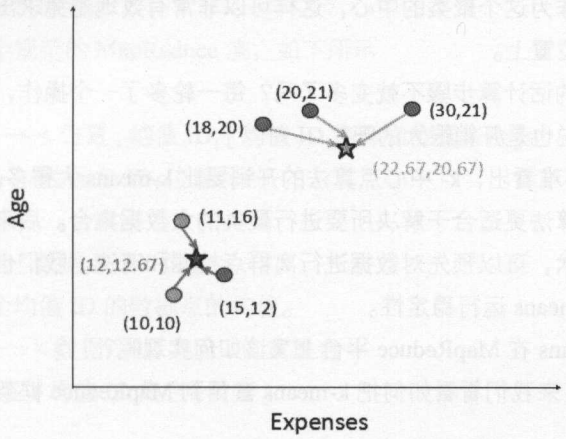
Mr. 王：别急，下面是算法的第 3 步。

第 3 步：计算已经在当前形成的类别的均值。所采用的方法就是求这一类别中所有的点在各个数据维度上的算术平均数，这样就可以得到一个新的均值。就像下面的图一样，左下角的五角星是左下角两个点的均值，右上角的五角星是右上角四个点的均值。此时的均值很有可能不再是某一个数据项。



第 4 步：现在我们又有了 k 个均值，再次让所有的点去计算其与哪个均值的距离更近。每一个点都会重新选择距离最近的一个均值，将其归入到该均值代表的类别中，此时新的类别划分就产生了。上图中的五角星，就是每一个点新发现的最近的均值。

重复执行第 3 步和第 4 步，直到类别的结果不再发生变化为止。



小可：也就是产生了不动点？

Mr. 王：是的。

小可：这回从直观上看结果好像已经不错了，相近的点都已经聚集在一起了。不过还有一个小问题，一开始我们随机选择均值，真的没有问题吗？

Mr. 王：你注意到了这个问题很好。其实可以证明，最初的均值随机选择并不会影响最终的聚类结果。但是并不意味它对算法是毫无影响的，如果所选择的点恰巧比较接近分类完成后的均值，那么算法收敛的速度就会变得快一些；如果所选择的点不够好，那么就会造成算法经过很多轮迭代才能收敛。也就是说，最初的均值随机选择，虽然不会影响算法的结果，但是会影响算法收敛的速度。

小可：怪不得要选择一些数据项作为初始点呢，这样比在空间中随机选择更接近聚类完成后的均值。

Mr. 王：不过这个算法也不是完美的，它也有缺陷。

小可：看起来得出的结果挺准确啊，哪里不好呢？

Mr. 王：在实际使用的数据中，非常有可能出现噪声或者离群点。大部分点都集中在某个区域里面，但是有几个点距离其他的点都非常远。你可以想一想，这样的点会使得我们求出的均值非常偏离大部分点所在的小区域，会造成聚类在一定程度上不准确。我们说 k-means 对于含有噪声和离群点的数据是不健壮的。

小可：那么怎么解决呢？

Mr. 王：后来计算科学家们提出了一种对 k-means 的改进算法，这种算法叫作 k- 中心点算法。

小可：哦，它和 k-means 有什么区别呢？

Mr. 王：它对 k-means 确定均值的方法做了一个小小的改进。k-means 的每一步直接采用每一个聚类中点的均值作为该聚类的中心；而 k- 中心点算法在求出了均值之后，会选择一个距离均值最近的数据项作为这个聚类的中心，这样可以非常有效地避免求出来的中心处在一个非常偏离大量数据点的位置上。

小可：嗯，这样的话计算步骤不就变多了吗？每一轮多了一个操作，而且确定这个中心点对于海量的数据点来说也是开销很大的吧？

Mr. 王：没错，不难看出，k- 中心点算法的开销要比 k-means 大得多。所以在实际应用中，我们就要权衡哪一种算法更适合于解决所要进行聚类的大数据集合。后来，科学家们也开发出了各种数据预处理技术，可以预先对数据进行离群点检测和清洗，我们也可以采用数据离群点清洗的方法来加强 k-means 运行稳定性。

小可：那么 k-means 在 MapReduce 平台上又该如何实现呢？

Mr. 王：好，接下来我们看看如何把 k-means 套用到 MapReduce 框架中。显然这也需要多轮迭代 MapReduce。

我们可以做如下的设计：

别忘了，在设计 MapReduce 算法时，首先要设计键值对。

数据点 ID → < 位置 , 均值 ID, [均值 ID 集合和位置集合]>

其中，数据点 ID 表示数据项编号，用于区分数据项，位置是该数据点在空间中的位置；均值 ID 是每一轮中已经确定的；后面的两个集合是目前已有的所有的均值 ID 及其位置。这里的均值 ID 可以用于最终标注分类的结果，比如可以称它们为第一类、第二类、第三类等。

于是，在 Map #1 中，输入数据点 ID → < 位置 , 均值 ID, [均值 ID 集合和位置集合]> 键值对每个数据点要去计算距离自己最近的均值，然后发出均值 ID → < 数据点 ID, 位置 > 键值对，这样就可以表示出一个在某特定位置、编号为数据点 ID 的点本轮被划归到具有某一个均值 ID 的类别中。

在 Reduce #1 中，Reducer 会收集到大量的均值 ID → < 数据点 ID, 位置 >，以便于重新计算属于一个均值 ID 的数据点的均值。这可以通过这些数据点的实际位置进行计算。然后输出为数据点 ID → < 数据点 ID 集合 , 位置集合 >，并且对于除了自己以外的那些均值 ID, 还要发出其他均值 ID →位置 (均值 ID 集合 , 位置)，以便于这些均值可以有效地知道其他均值所在的位置。

在 Map #2 中，直接将一轮 MapReduce 的值结果发送给 Reduce #2。

在 Reduce #2 中，对于在当前均值所代表的类别中的每一个节点，会输出数据点 ID → < 位置 , 均值 ID, [均值 ID 集合和位置集合]> 键值对。不难看出，它与 Map #1 的输入是一样的，

这样就可以让 MapReduce 进行一轮轮迭代，最后得出结论。同时，Reduce #2 还要输出数据点 ID →< 位置 , 均值 ID, [均值 ID 集合和位置集合]> 键值对。

如果 MapReduce 能够在这一轮中停止，那么后面输出的这一项将成为最终结果。

当整个过程达到不动点时，也就是在执行过程中各节点归属的类别不再发生变化时，整个算法执行完毕，系统停机即可。

上述过程形成一个规范的 MapReduce 流，如下所示

Map #1：

输入：数据点 ID →< 位置 , 均值 ID, [均值 ID 集合和位置集合]>

发出：均值 ID →< 数据点 ID, 位置 >

Reduce #1：

输入：均值 ID →< 数据点 ID, 位置 >

重新计算属于一个均值 ID 的数据点的均值。

输出：数据点 ID →< 数据点 ID 集合 , 位置集合 >

其他均值 ID →位置 (均值 ID 集合 , 位置)

Map #2：

直接将一轮 MapReduce 的值结果发送给 Reduce #2。

Reduce #2：

输入：Map #2 传递过来的数据。

输出：数据点 ID →< 位置 , 均值 ID, [均值 ID 集合和位置集合]> 键值对

这样我们就成功地通过 MapReduce 实现了 k-means 算法。在实际应用中，k-means 算法的输入数据量往往是非常大的，使用像 MapReduce 这种并行平台是非常常见的。

另外，当我们希望使用并行工具来解决一些数据挖掘问题时，可以使用一些现有的开源工具包，这样可以让我们的工作变得更加简捷。一个很有代表性的工具就是由 Apache 基金会推出的开源数据挖掘工具包 Mahout。Apache Mahout 基于 Apache Hadoop（Hadoop 是一个 MapReduce 开源实现版本），是一种可伸缩的数据挖掘工具包，可伸缩性可以非常有效地面对各种较大的数据规模。它可以帮助我们非常方便地完成频繁模式挖掘、分类和聚类的一些操作，其中有很多使用非常方便的 API，可以直接调用它们，使得数据挖掘工作变得轻松容易。

在 Apache Mahout 中，已经为我们预先实现了一些较好的聚类算法，包括 k-means、k-means 的模糊版本、Canopy、Mean Shift 等。当我们要进行一些简单的聚类时，可以直接使用这些组件包的库函数。

其实不论是 k-means 还是 k- 中心点算法在思想上都有一个小缺陷。K- 中心点虽然有效地降低了噪声和离群点对算法的影响，但是依然没有解决一个问题，就是像 k-means 这样的 k-cluster 问题，也就是都需要先指定一个 k 出来。所以在有些情况下，指定这个 k 值可能会成

为一个麻烦。如果大量的数据分布非常的密集、杂乱，很难从直观上看出这些大量杂乱的点应该分成几类时，我们所指定的不准确的 k 值也有可能影响聚类结果。

小可：的确，k-means 的每一轮都要算出 k 个均值，如果不知道 k 是多少，那就无法运行了。如果给出一个不准确的 k，又不能最好地代表数据的分布情况。

Mr. 王：所以 k-means 也不是一种万能的聚类方法。至于对这种问题的解决，科学家们提出了基于密度的聚类方法，在这里我就不展开谈了。如果你有兴趣，可以去查阅一些关于 DBSCAN 算法的书籍和论文，相信会很有帮助的。

9.4 分类算法——Naive Bayes

小可：说完了聚类，那么分类算法又是怎么做的呢？

Mr. 王：我们知道，分类是首先通过对训练集中大量数据的分析，训练出一个分类的模型，或者说得出一个分类的标准，然后使用这个标准对后面再到来的数据进行分类。所以我们的大部分工作都集中在对训练集的处理上。

这里介绍一种经典的分类算法——朴素贝叶斯分类器（Naive Bayes）。这种分类方法非常简单，但是非常有效。

小可：我在学概率论时听说过贝叶斯定理，和这个是一个道理吗？

Mr. 王：朴素贝叶斯分类器依据的核心原理就是贝叶斯定理。你还记得贝叶斯定理吗？

小可：假设有两个事件 A 和 B，它们之间具有这样的关系：$P(B \mid A)=\dfrac{P(A \mid B)P(B)}{P(A)}$。可是贝叶斯公式看起来也不像和分类有关系啊。

Mr. 王：现在我们就来谈谈贝叶斯公式是如何有效地运用在分类算法中，并形成了非常著名的贝叶斯分类器的。其实所有的分类方法都是希望经过训练后，模型可以输出某条数据记录最可能的分类，也就是通过训练集 T 的训练，找到一条记录 r_i 最可能的分类 c_j。而“最可能”这个概念用概率来描述和衡量是非常合适的。我们可以将上面描述的分类算法所要解决的问题，表示为概率 $P(c_i|r_j)$，它的意义就是在数据项为 r 的条件下，其分类是 c 的概率。我们希望找到的就是：

$$c_{\text{result}} = \arg\max P(c_i \mid r_j)$$

即，使得 $P(c_i|r_j)$ 最大的 c_i。在这种思想的支持下，也就有了以朴素贝叶斯分类器为代表的贝叶斯分类方法。

小可：哦！这样的表述的确很有道理啊，将寻找最可能的分类，转换为使用概率的方法。那么根据贝叶斯定理，就可以把 $P(c_i|r_j)$ 表示成：

$$P(c_i \mid r_j) = \frac{P(r_j \mid c_i)P(c_i)}{P(r_j)}$$

Mr. 王：很好，这其实就是朴素贝叶斯分类器使用的核心方法。我们简单解释一下各个量的意义。$P(c_i)$ 表示 c_i 这个分类的先验概率，就是我们在遇到训练集之前，对 c_i 这个分类的先验认识。这个先验概率来自于我们对 c_i 的预先研究或统计结果，比如我们要通过贝叶斯方法确定一个人是不是患有疾病 x，如果通过统计知道人群中只有 1% 的人有这种疾病，那么 $P(c_x)$=0.01，而 $P(c_{\text{non-x}})$=0.99。

而$P(r_j \mid c_i)$是假设在 c_i 成立的条件下，r_j 出现的概率。这部分证据主要来自于训练集，观察训练集中 r_j 的各种属性与 c_i 的关联程度。这是在下面的计算中我们要重点讨论的部分。

小可：那 r_j 呢?

Mr. 王：其实不难发现，$P(r_j)$ 是一个和 c_i 无关的量。虽然它和计算 $P(c_i|r_j)$ 有着直接的关系，但是当我们要进行分类时，r_j 已经确定，所以在实际的计算中无须计算它。不过这个量并不是完全没有意义的，r_j 反映了这种特征的数据本身出现的概率。在贝叶斯公式的分子一定的情况下，如果 $P(r_j)$ 很大，则说明它出现的概率本身就很高，而不论是不是在分类 c_i 中。从直观上看，这就不是一个很好的分类特征，比如说文本分类中的 is、am、are，这样的词在所有的样本中出现得都很多，却与我们是不是对这篇文章感兴趣基本没什么关系。

现在我们通过一个实际的例子，看看贝叶斯分类器是如何工作的。

比如有关于一批书籍的记录，表示为：

< 主题 s，语言 l，难度 d，厚度 t>

1 < 计算机，中文，易，薄 > 喜欢

2< 计算机，中文，难，薄 > 喜欢

3 < 计算机，中文，难，薄 > 喜欢

4 < 计算机，中文，难，厚 > 不喜欢

5 < 计算机，英文，易，厚 > 不喜欢

6 < 电子，中文，易，薄 > 喜欢

7 < 电子，中文，易，厚 > 喜欢

8 < 电子，中文，难，厚 > 不喜欢

9 < 电子，英文，易，厚 > 不喜欢

10 < 电子，英文，易，薄 > 不喜欢

这些书籍分别按照主题、语言、难度、厚度进行了特征标注，并且标注了读者 A 是否喜欢这本书。现在有了一本书 11 < 计算机，中文，易，厚 >，我们希望知道读者 A 是否会喜欢这本书，这时就可以运用朴素贝叶斯分类器。首先想一想，我们希望得到的是什么?

小可：我想可以用这个式子表示吧?

$$c_{result} = \text{argmax}P(c_i|s=\text{CS},l=\text{Chinese},d=\text{easy},t=\text{thick})$$

Mr. 王：很好，那么现在我们可以分别去求解：$P(\text{like}|s=\text{CS},l=\text{Chinese},d=\text{easy},t=\text{thick})$ 和 $P(\text{dislike}|s=\text{CS},l=\text{Chinese},d=\text{easy},t=\text{thick})$，然后比较哪一个更大就可以了。

根据贝叶斯定理进行展开，就可以转而求解（这里可以忽略分母，它并不影响最终的分类结果）：

$$P(s=\text{CS},l=\text{Chinese},d=\text{easy},t=\text{thick}|\text{like})P(\text{like})$$

和

$$P(s=\text{CS},l=\text{Chinese},d=\text{easy},t=\text{thick}|\text{dislike})P(\text{dislike})$$

小可：那么如何去求解 $P(s=\text{CS},l=\text{Chinese},d=\text{easy},t=\text{thick}|\text{like})$ 呢？训练集中并没有这样的样本？

Mr. 王：其实这个问题可以解释一个小疑惑，就是朴素贝叶斯分类器为什么被称作“朴素的”贝叶斯分类器。这是因为它做了一个假设，就是某一个元组中一个属性的值对它最终属于哪一个类别的影响与其他属性值是相互独立的。每一个值的取值都独立地影响它属于的类别，之间不相关。这一假设被称为类条件独立性。而在实际生活中，这个并不是完全准确的，属性之间往往存在着一些联系。由于这个原因，我们叫它“朴素的”贝叶斯分类器。但在实际应用中，朴素贝叶斯分类器的准确率还是非常高的，可以和一些非常复杂的模型相媲美。

如果两个事件是条件独立的，那么就有乘法原理：

$$P(AB) = P(A)P(B)$$

所以：

$P(s=\text{CS},l=\text{Chinese},d=\text{easy},t=\text{thick}|\text{like})=P(\text{CS}|\text{like})P(\text{Chinese}|\text{like})P(\text{easy}|\text{like})P(\text{thick}|\text{like})$。

小可：这样就可以通过 like 中 CS 的计数来求解 $P(\text{CS}|\text{like})$ 了。只要统计出在训练集中被标注为 like 的书籍中 CS 的计数，然后与 like 的计数做商即可。果然计算变得非常容易了。

Mr. 王：依此类推，分别求出 $P(\text{CS}|\text{like})$、$P(\text{Chinese}|\text{like})$、$P(\text{easy}|\text{like})$，$P(\text{thick}|\text{like})$，就可以求出 $P(\text{CS,Chinese,easy,thick}|\text{like})$。也就是说，朴素贝叶斯分类器认为：

$$P(r_j|c_i)=P(a_1|c_i)P(a_2|c_i)P(a_i|c_i)\cdots P(a_n|c_i)$$

其中，$a_1 \cdots a_n$ 为 r_j 的 n 个属性。

Mr. 王：接下来我们还需要求出先验概率 $P(\text{like})$。在这个例子中，只要通过对训练集中 like 和 dislike 的计数来计算 $P(\text{like})$ 就可以了。在其他的问题中，可以有很多不同的办法来确定先验概率，如果实在缺乏相关的先验知识，我们可以认为所有分类的概率都相等。

现在我们来看看如何用 MapReduce 框架来完成一个贝叶斯分类器的训练和搭建。

在第一轮 MapReduce 的 Map 阶段，我们输入的键值对就是 [记录编号 → < 类别，特征 >]。然后 Map 会向外发出 < 特征，类别 > → 1 这样的键值对。

在 Reducer 中，我们求出形如 < 特征，类别 > →数量这样的键值对。

小可：这个跟 word count 好像啊。

Mr. 王：本质上这个步骤就是一个 word count 的操作。其实贝叶斯公式的训练过程，就是对样本中各种特征和类别进行统计的过程，其基本操作依赖的就是每一种特征和类别关联之后的数量，使用的方法其实就是 word count。

在第二轮 MapReduce 中，我们要求解整个训练集中每一种特征一共有多少个元组，以方便求解各个概率。其实这个过程也是一个 word count 的操作。只要让所有的 Mapper 输入的记录编号 → < 类别 , 特征 > 这样的元组，每收到一个记录编号，就发送 1，然后在 Reducer 中，将收到的这些 1 统计起来，得到一个特征→计数即可。

MapReduce 的算法框架如下所示。

Map #1：

输入：记录编号 → < 类别 , 特征 >

输出：< 特征 , 类别 > → 1

Reduce #1：

输出：< 特征 , 类别 > → 计数

Map #2：

输入：记录编号 → < 类别 , 特征 >

输出：特征 → 1

Reduce #2：

输出：特征 → 计数

Mr. 王：最后，我来简单总结一下分类和聚类这两类算法在大数据并行平台上的一些特点。

在聚类中，一般算法都会经过多轮迭代或者处理步骤。比如在典型的 k-means 算法中，我们要不断地重新去发现每个集合的均值，然后重新计算所有的点与这些均值的距离，再重新归类。最后我们还要通过不动点的判定，来确定整个平台是不是达到了一个收敛的状态。

而分类算法往往是比较复杂的，我们选择了非常经典的朴素贝叶斯分类器，好在它的处理相对比较简单。由于分类是一种基于训练集的学习再对新来的元组进行分类的一种方法，我们要做的就是去计算训练集的概率分布，并且用它去估计客观世界中的概率分布。所以，分类算法是一个计算概率分布的过程。一般来说，分类算法常常会涉及条件概率的求解。而概率的求解，我们可以通过计数的方法，体现在 MapReduce 中，可以用一轮 MapReduce 对分子计数，另一轮对分母计数，从而通过两次计数求出一个概率的值。

在 Apache Mahout 中，也有分类算法的实现，Mahout 的内部直接包含有一个 Naive Bayes 分类器的示例程序，如果感兴趣的话，不妨去试着运行一下它。不过要记住，Mahout 是一个基于 Hadoop 的工具包，如果想要运行 Mahout，就要先安装和配置一个 Hadoop MapReduce 平台。我们会在后面的章节中介绍 Hadoop 平台的配置和使用。

第10章 推荐系统

10.1 推荐系统概述

Mr. 王：在现在的网络购物平台和电影、图书分享平台中，还有一类非常广泛存在的机制——推荐机制。

小可：在我打开的购物网站右侧经常有一列推荐购买的商品，它们往往和我以前购买过的商品非常相似，或者是同一个品牌，或者是同一种类型的商品，或者是广受大家欢迎的商品。

Mr. 王：今天我们就来讨论关于推荐系统的问题。在很久以前，对于零售商来说，货架上的商品可能只有少数几种，当用户提出一个需求时，零售商将商品全部拿出来放在桌面上供用户选择也是非常容易的。但随着互联网逐渐发达起来，以及各种网络购物平台的兴起，商品不再是一种稀缺的资源，网络使零成本产品信息传播成为可能。这也就导致了商品信息在网络上过剩的情况。

小可：是啊，现在购物平台上的商品真是太丰富了，每当我打开一个购物网站，搜索一个关键词时，就会有超过10页的内容被检索出来。

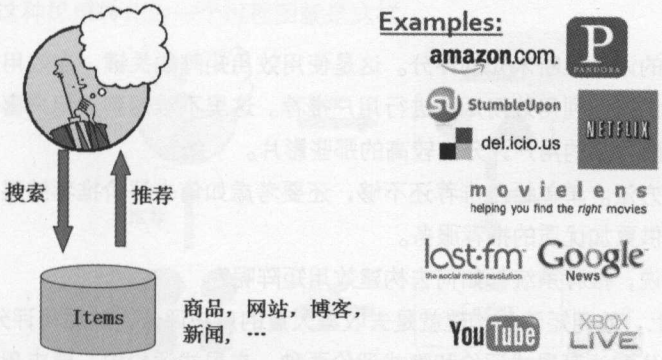
Mr. 王：不过经常网购的人会知道，第二页以后的内容，往往就很少去看了。

小可：是啊，多数人是不会把找出来的10页内容都看完的，太浪费时间了，大多数时候我只看第一页。

Mr. 王：不仅仅是在网络购物平台上，在电影和书籍这样的交流平台上也是如此。现在网络上的影片非常多，书籍也非常多，当我去搜索一个类别时，比如输入“科幻，美国，1990年以后”这样的关键词之后，也会产生大量的检索结果。此时也存在影片或者书籍信息过剩的

问题。包括在 XBOX 和 STEAM 这样的游戏平台上，将哪些游戏推荐给用户；在 ZAKER、今日头条这样的新闻资讯平台中，选择将哪些消息推送给用户；在 YouTube 这样的视频网站中，如何去发现用户喜欢的视频，并推荐给用户，都是平台不得不考虑的问题。

就如你说的一样，在平台向用户推送了大量的信息之后，多数人会去看的信息只有前几页甚至只有第一页，他们不会看完所有的检索结果。现在我们就要考虑一个问题，既然用户并不关心所有的检索结果，而只关心放在最前面的那些结果，那么一个购物平台的设计者就要研究应该将哪些内容放在最前面；如果采用只返回检索结果的一部分的策略，要考虑的就是将哪一部分返回给用户。所以在这些平台中，需要设计一个“推荐系统”。



小可：的确，各大网站和平台都有自己排序机制，这都是推荐系统在发挥作用吧。

Mr. 王：从用户角度来看，你见过哪些推荐系统呢？

小可：隐性的有比如检索结果的排序；显性的就像淘宝网右侧的推荐列表、要目列表；音乐软件中常见的有热歌 TOP 100、最流行、最新上载；还有像 Amazon、京东这样的网站会根据用户最近的访问情况和之前的购买情况，对每个用户进行个性化定制，推荐用户可能会喜欢的商品。

Mr. 王：好，接下来我们就来研究这些推荐系统的原理，看看这些推荐系统内部使用了怎样的方法，从而实现高准确度的推荐。

首先我们要看看一个推荐系统的模型是怎样的。

对于一个推荐系统来说，两个基本的要素就是用户集 X 和项目集 S。在一个电影推荐平台中，X 就是访问网站请求推荐，并且对电影进行评分的那些用户的集合；S 就是影片的集合。

于是我们给出一个评估模型，称作效用矩阵。在效用矩阵中，每一行是一个用户，每一列是一个项目，每一个数据记录着某个用户 $x \in X$ 对某个项目 $s \in S$ 的一个评分。比如下图：

基于效用矩阵，我们提出效用函数的概念。在不同的系统中，效用函数的定义可能不完全相同，但其都符合效用函数 $u: X \times S \rightarrow R$ 这种形式。其中 R 是评分集，它是一个完全有序集。常见的 R 就比如在电影评分中那种 0 ~ 5 星的形式，或者简单一点，就是 [0,1] 之间的一个实数等。

	Avatar	LOTR	Matrix	Pirates
Alice	1		0.8	
Bob		0.5		0.3
Carol	0.9		1	0.8
David			1	0.4

形成并使用效用矩阵模型的主要步骤如下：

（1）收集已知评分形成效用矩阵。在实际应用中，就涉及如何去收集这个效用矩阵中数据的问题。

（2）根据已知的评分推断未知的评分。这是使用效用矩阵的关键，当效用矩阵中的数据已经收集好之后，我们如何利用效用矩阵进行用户推荐。这里不难想到，用户喜欢的影片应该是自己没有评分过，而且别的用户评分比较高的那些影片。

（3）评估推断方法。单单给出推荐还不够，还要考虑如何去评价推荐结果，以便能够改进推荐系统，从而提供更加优质的推荐服务。

小可：一般来说，推荐系统都如何去构建效用矩阵呢？

Mr. 王：本质上，效用矩阵的构建就是去收集大量的用户评分，然后将评分填到矩阵中去。常用的收集用户评分方法有显式评价和隐式评价两种。在显式评价中，要求用户直接对项目给出评分。这种方法在实际中并不总受用户欢迎，有些用户比较喜欢评分和写影评工作，而有些用户会觉得进行评分是一项很麻烦、累赘的工作，对自己是一种困扰。而在隐式评价中，不直接向用户索求评分，而是从用户的行为中学习其评分。比如在网络购物中，就可以根据用户以往的购买习惯，推测用户喜欢什么样的商品，从而给出推荐。就是基于这样一种思想，用户购买意味着用户对这件商品给出了较高的评分。当然这不是绝对的，用户不购买就代表低评分吗？这似乎也不一定，所以在这样的应用环境中，“什么代表低评分”也是一个值得思考的问题。

小可：嗯，很多用户确实不太喜欢对他们所选择的每一个项目都进行评分，在音乐平台上，如果我每听一首歌系统都要提示我给出一个评分的话，也确实挺麻烦的。

Mr. 王：正是由于用户有这样的看法，导致大多数人没有评价过大多数项目，换句话说，大多数项目是没有人进行评价的。这就导致了效用矩阵 U 往往是非常稀疏的。另外，推荐系统还存在“冷启动”的问题。一些新推出的项目是没有评分的，它们刚刚出现在平台上，用户还没来得及对它们进行评分；相应的，平台对于新用户的喜好也是认识不足的，因为平台中没有任何关于新用户的历史数据。这也是两个比较棘手的问题。

小可：嗯，这些问题在实际的系统中的确非常常见，那我们该如何去解决这些问题呢？

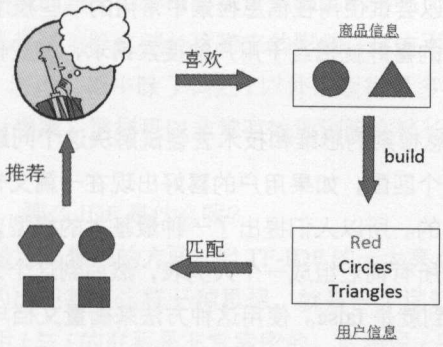
Mr. 王：接下来就介绍几种在实际中常常用到的推荐方法，看看它们是如何有效应对这些问题的。

10.2　基于内容的推荐方法

Mr. 王：最常见的一种方法就是基于内容的推荐。基于内容的推荐思想非常的清晰、简单，就是向用户推荐与他评分高（喜欢）项目相类似的项目。

小可：嗯，很多电影网站根据用户的喜好进行定制，使用的都是这种策略。还有就是很多音乐平台都有一个栏目叫作“猜你喜欢”，这个栏目就是根据用户收藏的那些“我喜欢”的歌曲或者下载的歌曲进行推测的吧。还有一些博客网站也会向用户推荐与其浏览记录比较相似的网站。

Mr. 王：将这种思想转化为一个过程图就是这样：



用户在使用某个平台的过程中表示比较喜欢某些商品，系统就会首先记录下该用户比较喜欢的那些商品信息，然后通过这些商品信息去建立一个用户信息库，这个信息库包含的是用户喜欢的那些商品所具有的一些特征。比如在上面这个例子中，用户表示比较喜欢红色的圆和红色的三角。系统记录下这些信息之后，就会进行分析处理，得出的结论是，用户喜欢红色的东西，喜欢三角形和圆形的东西，然后就会到系统的数据库中去检索符合这些特征的项目，并将其推荐给用户。

在具体的实现上，它使用的是一种叫作项模型的方法。简而言之，就是对每一个项目建立一份 item profile，即项目简介或者项目概括。不过在具体的系统中，这种概括不能太随意，而是要包含我们可以进行分辨的那些标准化的特征。

小可：就像电影的片名、导演、分类、主要演员等，一首音乐的演唱者、作词作曲家、流派、风格，甚至适合的场景等。

Mr. 王：嗯，这些都是比较常见的特征。另外值得注意的是，在文本检索中，很多文本本身并不具有很明显的特征，我们经常需要在文章内自行提取关键词来进行标注。对于像科研论文这样的文章或许会好得多，因为文章中有关键词这一栏。但是对于那些非文本类的数据，就不那么容易了。比如对于图片的推荐，就会由于特征难以由计算机进行提取抽象而实现起来比

较困难。想想看，有什么好办法吗？

小可：前面我们不是讲过众包算法吗？我认为可以把图片库在网上众包出去，然后留下一些问题，让用户进行回答，通过这些问答来获得我们关心的特征。

Mr. 王：很好，由人工对那些非文本数据进行标注是一种常用的策略，体现了众包算法的思想。

小可：如果没有人参与到对项目进行特征标注的工作中，那么一般基于内容的推荐要怎么实现呢？

Mr. 王：这里我用文档进行举例，谈谈如何对一个文档库中的文档进行推荐。其实在文档库中按照用户的喜好去推荐文档这项工作，和在一个文档库中按照用户的搜索请求去搜索文档是非常相似的，所以我们可以尝试使用在信息检索中常用的一些技术。

小可：的确，其实用户的喜好就相当于用户的搜索请求，和我使用像百度文库和谷歌学术这样的网站挺像的。

Mr. 王：现在我们用信息检索的思维和技术去尝试解决这个问题。其实这个工作就是对用户的喜好和文档的内容做一个匹配，如果用户的喜好出现在一篇文档里，我们就倾向于认为这篇文档和用户的喜好是相关的。所以人们提出了一种最基本的模型，叫作布尔模型。所谓布尔模型，就是将文档中出现的所有词汇组成一个大列表，然后到这个大列表中去找用户的喜好词汇，找到了就是 true，没找到就是 false。使用这种方法来衡量文档与用户喜好的关系。

不过人们渐渐发现了这种模型的问题：在文档中出现过一个词汇，并不能代表这篇文档跟这个词汇的关系是比较大的。比如，这个词汇在这篇文档中可能只是简单地提到了一次，而另一个词汇却是文档中重点讨论的内容，它们与文档内容的关联程度显然是不同的。不过对布尔模型来说，这两个词汇都是“在文档中出现过”的。

小可：嗯，有些词汇在文档中虽然出现了，但是不重要；而有些词汇却是反复出现且很重要的，这些词汇就是所谓的关键词吧？

Mr. 王：很好，现在问题的关键就可以转化为如何确定文档中的关键词。在信息检索技术中有一种比较成熟的技术叫作“向量空间模型”，它使用一种关键词衡量指标“TF-IDF”。

为了发现关键词，我们可以对文档中的所有词汇进行评分，评分比较高的词汇就是关键词。想一想，这个评分与什么有关？或者说，什么样的词汇更倾向于成为代表一篇文档内容的关键词呢？

小可：我觉得词汇出现得多，相对来讲就应该比较关键吧。

Mr. 王：这就是 TF-IDF 的第一个组成部分：TF，也就是词频（Term Frequency）。在向量空间模型中，设 i 为一个关键词，j 为一篇文档，我们首先关注的就是 TF(i,j)，也就是在 j 文档中，i 的词频。

它可以表示为：

$$\mathrm{TF}(i,j)=\frac{\mathrm{freq}(i,j)}{\max \mathrm{others}(i,j)}$$

其中，freq(i,j) 就是 i 在 j 中出现的频率，而 maxothers 就表示 i 除了 j 以外最大的 freq(x,j)。

小可：可是一般来说，只要有 freq(i,j) 这个量就可以衡量一个词汇在一篇文档中出现的频率是不是比较高了吗？为什么还需要考虑 maxothers 呢？

Mr. 王：这是 TF 这个量设计得比较巧妙的地方。文档库中的文档显然长度是不一样的，有时文档的长度相差还比较大，这就引发了一个问题：在较长的文档中，包含某个关键词的概率就更大，或者说更容易包含某些关键词。也就是说，长文档在包含关键词方面比短文档拥有比较明显的优势。而我们期待的是文档本身与这个关键词的关系比较大，而不是由于其长度的原因，所以需要一些策略来尝试消除文档长度带来的影响。这方面的方法也是有很多的，其中就包括在 TF 值中使用的，考虑文档中除了词汇 i 以外出现得最多的词汇的词频是多少，通过两个绝对频率求出一个相对频率，这样可以非常有效地消除文档长度带来的影响，对文档长度进行有效的归一化。

小可：嗯，知道了 TF，那么 IDF 是什么呢？

Mr. 王：词频是一个比较容易想到的方面。但 TF-IDF 的一大亮点是它考虑到了 IDF 这个量。IDF 也叫反文档频率，它的出现基于这样一种思想，就是一个词汇 i 如果出现在 j 中，在其他文档中都没有出现，则说明 i 与 j 的联系是非常紧密的，或者说 i 是 j 的关键词。而若 z 在 j 中出现，也在其他文档中出现，即 z 在所有文档中都出现得比较多的话，我们就不认为它是一个与 j 相关的关键词，只能说明 z 是一个常见的词汇。

小可：就比如 the、a、is、are 这样的？

Mr. 王：是的。所以我们引入 IDF 来降低那些在所有文档中都会出现的关键词的评分。设 i 为某词汇，N 为整个文档库中的文档数量，$n(i)$ 为词汇 i 在其中出现过的文档数量，则 i 的 IDF 定义为：

$$\mathrm{IDF}(i)=\log\frac{N}{n(i)}$$

小可：嗯，出现过 i 的文档越多，IDF(i) 就越小，就会降低它和 j 之间的关系。

Mr. 王：综合 TF 值和 IDF 值，我们可以给出 TF-IDF 模型：

$$\mathrm{TFIDF}(i,j)=\mathrm{TF}(i,j)\cdot\mathrm{IDF}(i)$$

这样每篇文档就可以表示成其所包含的词汇与其 TF-IDF 值的一个向量。

小可：嗯，这种模型看起来比布尔模型对一个文档中包含的关键词的衡量更加准确一些，但是一篇文档往往是很长的，这个向量岂不是会很大吗？

Mr. 王：这是向量空间模型的局限性之一。人们为改善这种模型也做了很多的努力，比如

引入停用词，像 the、a 这样的词汇，在每篇文档中都出现得非常多，而且还完全无法表达一篇文档的实际意义，没有任何价值。我们就要将这样的词汇列为停用词，不参与 TF-IDF 值的计算，也就不会出现在向量中了。另外，在英语这样的语言中，词根和时态变化是非常多的，如果单纯地将它们作为文本分析的话，就会将它们视作不同词汇，也就会导致有大量表示相同意义的词汇被当作了向量的不同维度，这是很浪费空间的。于是，人们提出了词干还原的方法，对词汇的各种语气时态进行归一化处理，比如 write 和 wrote 就会被视作一个词汇，这样就可以有效地缩减向量的大小。

这方面的技术也有很多，如果感兴趣的话，可以去查找一些信息检索方面的书籍。

Mr. 王：基于内容的推荐方法还是有很多优点的。它不需要其他用户的数据，也就是只与自己的喜好和项目的特征有关。由于不需要关注其他用户的评分，也就没有冷启动或者效用矩阵稀疏性问题，新项目或不流行项目推荐虽然没有其他用户对其进行有效的评分，但是却可以因为其所具有的特征与用户喜欢的项目特征进行匹配，从而被推荐系统提取出来。另外，对于品味一致的用户，我们可以做相似的推荐，不需要为每个用户都重新运行一次推荐算法，这样比较方便。而且基于内容的方法能够有效地为自己做出的推荐提供一个合理的解释，也就是对推荐项目给出对应的内容特征描述，而不是依赖于其他用户的评价。

不过基于内容的推荐也不是十全十美的。前面我们也提到过，对于一些非文本数据，找到适当的特征是困难的。比如对于图片、电影和音乐来说，发现它们的特征还是具有一定难度的，而使用众包方法会带来额外的成本和时间开销。

小可：而且这些特征对原影片和音乐未必能做到完整而准确的概括，即使流派和作者都一样，不同的歌曲在某些难以用特征来描述的方面也可能相差很大。

Mr. 王：另外，基于内容的推荐还存在过度集中的问题。当我们让用户来使用这个系统时，只会询问几个简单的问题来了解用户的偏好，如果询问过于详细反而降低了用户使用系统的兴趣；但由于得到的喜好特征比较少，就会出现查询出来的内容非常集中，它们的相似度过高没有多样性，也就不会向用户推荐内容偏好模型之外的项目了。

小可：嗯，人们可能有多方面的兴趣，推荐出来的内容太过相似难免有一种重复的感觉。

Mr. 王：而且由于我们推荐的内容完全基于用户的喜好特征，不能利用其他用户的优质判断，这一点既给系统带来了优势，也自然造成了一些问题。

10.3 协同过滤模型

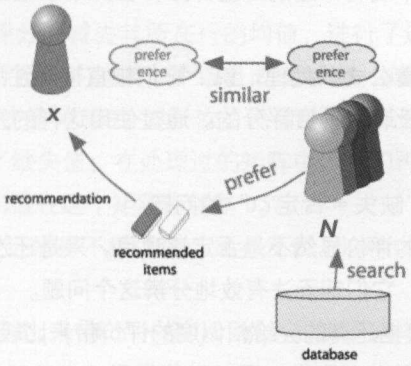
Mr. 王：为了能够有效地利用其他用户的评价来进行更有效的推荐，人们提出了协同过滤的推荐模型。

小可：那什么是协同过滤模型呢？它又有哪些优势呢？

Mr. 王：先说说协同过滤的思想。既然认为他人对一个项目的评价是有一定价值的，我们就要尝试去利用他人对一个项目的评分来考察该项目的好坏。但是这里存在一个问题，就是他人认为不好的项目不一定是我们认为不好的，或者说他人的评价标准不一定符合我们的评价标准。我们要去考虑，如何能够让那些和某个用户评价标准相似的人来评价该用户没有评价过的东西。这样，给出的评价就会更加符合该用户的喜好，从而推荐质量就会提高。

小可：确实是这样，喜好和我不一样的人评价好的东西不一定是我喜欢的。可是网络上的人有很多，我如何去发现那些喜好和我一样的人呢？

Mr. 王：我们将这种思路形式化，协同过滤就是当我们研究用户 x 时，去找到与 x 有相似评分的用户集合 N，根据 N 中用户的评分估计 x 的评分。



小可：原来是比较两个人之间的评分啊。嗯，如果我们两个人看过相同的几部电影，而且对这几部电影的评价相似或者一致的话，我们的确有可能有相似的喜好。

Mr. 王：如果你们的喜好相似，对新项目的评价很可能也是相似的。这是协同过滤依照的一个基本假设。那么现在的问题就归结为，我们如何去发现相似的用户呢？当用户数目不多，而且评价过的项目并没有那么多时，可能会出现几个人对这几个不多的项目做出完全相同的评分；但是当项目足够多时，很难有完全相同的评价，我们就要尝试使用一些标准来判断哪些用户过去做出的评价是足够相似的。

我们以评分为例，令 r_x 为用户 x 的评分矢量，比如以星级评价就可以表示成：

$$r_x = [*, \_, \_, *, ***]$$
$$r_y = [*, \_, **, **, \_]$$

其中的下画线就表示没有做出评价。在实际的系统中，做出这样评价的可能性是非常高的。

小可：因为非常容易出现两个人都没有看过的电影。

Mr. 王：我们可以尝试使用数学中的各种相似度函数，比如 Jaccard 相似度、余弦相似度或者皮尔森相关系数等。

比如余弦相似度，我们就可以将评分矢量进行标准化，转化成数字：

r_x, r_y **as points:**
$r_x = \{1, 0, 0, 1, 3\}$
$r_y = \{1, 0, 2, 2, 0\}$

然后使用余弦相似度的数学公式即可：

$$\text{sim}(x, y) = \cos(r_x, r_y) = \frac{r_x \cdot r_y}{||r_x|| \cdot ||r_y||}$$

余弦相似度是一种非常朴素的处理方法，处理起来也非常方便，但它也有其缺陷。不知你注意到没有，在原特征向量中没有标注的那些评分，在开始余弦相似度判定之前的标准化是如何处理的。

小可仔细思考着，观察着公式的处理，说：它们被直接标注成了 0，在公式中是无法区分这些用户给了项目低分还是没有对项目评分的。通过使用这样的方法，会直接认为他们都不喜欢这个项目。

Mr. 王：这就涉及一个“缺失 = 否定？”的问题。

小可：缺失对一个项目的评价显然不是否定该项目，只是还没有来得及评价罢了。而直接将向量交给公式去计算的话，它们却无法有效地分辨这个问题。

Mr. 王：有时，缺失的数据还真的会给相似度的评价带来比较大的干扰。

	HP1	HP2	HP3	TW	SW1	SW2	SW3
A	4			5	1		
B	5	5	4				
C				2	4	5	
D		3					3

Mr. 王：这是一个效用矩阵的例子，左侧的 ABCD 代表用户，上面的一排代表项目名称，中间的数据域是用户对项目的评分，我们给出的取值范围是 1 ~ 5，也就是网络上常用的 1 ~ 5 星的评价。

小可：这个矩阵里还真是有大量的缺失值。

Mr. 王：这些缺失值的存在就严重地干扰了我们对相似度的评价。从直观上看，我们会觉得 A 和 B 更相似一些，他们虽然只共同看过一部电影 HP1，但是评分非常接近（4 和 5）。而 A 和 C 共同看过的电影有 TW 和 SW1，不难看出，这两个用户对这两部电影的评价是完全相反的，一个接近最高分，另一个接近最低分，这说明他们对电影的喜好评价是不一样的。但如果通过前面提出的判据进行数值计算上的比较，就会发现一些问题。

Mr. 王在计算器上按了几下，说：如果用 Jaccard 相似度进行衡量的话，A 和 B 的相似度仅有 1/5，而 A 和 C 的相似度却有 2/4。这说明如果推荐系统用 Jaccard 相似度进行判定的话，反而会认为 A 和 C 相似。如果用余弦相似度进行判定的话，sim(A,B) 和 sim(A,C) 分别为 0.386

和 0.322，虽然判定为 A 和 B 更为相似一些，但是两者的计算结果非常的相似，这意味着如果没有 AB 这组作为对照的话，从数值上看我们会认为 AC 这组是非常相似的。从这种结果来看，由于这些缺失值的存在，我们的判据都出现了不同程度的失准。

小可：麻烦应该就出在这些缺失值上吧，我们的方法将它们看作了 0 分。这的确不够公平。

Mr. 王：为了解决这种问题，人们提出了中心化方法。来看看下面这个表格：

	HP1	HP2	HP3	TW	SW1	SW2	SW3
A	2/3			5/3	-7/3		
B	1/3	1/3	-2/3				
C				-5/3	1/3	4/3	
D		0					0

我们让表格中的每一个评分都减去其所在行的均值，进行了这样的处理之后，不难发现，所有的值都变成了关于自己评价的均值的一个增量或者减量。接下来，再次计算 sim (A,B) 结果就是 0.092，sim(A,C) 结果就是 -0.559。这一次，两者的差距就变得大多了。

小可：嗯，这样做忽略了缺失值，在处理过的矩阵中，我们再将其视为一个最低分，而将其中心化成一个均值。因为均值在这个矩阵中就是 0。

Mr. 王：如果你的概率统计学得不错的话，还会发现，这种以 0 为中心的数据求解的余弦相似度就是它们的相关系数。

小可在纸上写下了几个公式，计算了一会儿，说：的确是这样啊。

Mr. 王：有了前面这些准备工作，于是我们就可以根据这些相似的用户，去对一些没有进行评分的项目进行评分，并且将预测评分较高的那些项目推荐给用户了。

设 r_x 为用户 x 的评分矢量，N 为对其他的评分与用户 x 最相似的 k 个用户的集合。那么用户 x 对其没有进行过评分的项目 i 的评分预测为：

$$r_{xi}=\frac{1}{k}\sum_{y\in N} r_{yi}$$

小可：用与我较相似的人对这个项目的评分的平均数预测我的评分，还算合理吧。

Mr. 王：还算合理？你有什么更好的想法吗？

小可仔细思考着，说：我觉得虽然 N 中的人都是和我很相似的，但是其中依然有些人和我不那么相似。有时候由于 k 取得很大，导致 N 里面的人也比较多，最后找不到和我很相似的人，就会找一些不那么相似的人以求填充集合 N。而他们的评分会干扰到对我的估计。所以我认为，应该加权，也就是说，与我更相似的人给出的评分，应该对预测我的评分影响更大一些。

Mr. 王满意地说：非常好，前人也考虑到了这个问题，于是就采用了相似度的加权方法，写成表达式就是这样：

$$r_{xi}=\frac{\sum_{y\in N} s_{ij}\cdot r_{xj}}{\sum_{j\in N} S_{xy}}$$

在这个式子中，就做到了和你更相似的人的评分在对你的预测中会占更大的比重。式中，s_{xy} 就表示 x 和 y 两个用户的相似度。

Mr. 王：在实际应用中，我们不仅要考虑用户与用户的相似度，还要考虑项目和项目的相似度。对于每一个项目 i，我们都去寻找其他相似的项目，根据相似项目的评分估计项目 i 的评分。

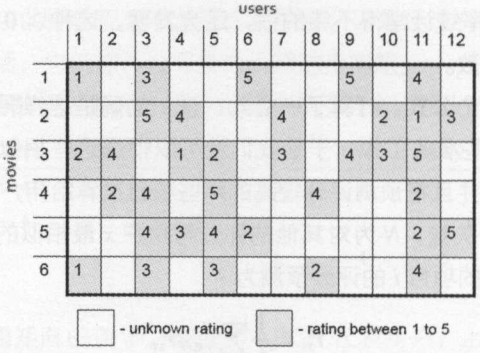
小可：看起来和用户相似度的思想很相像啊，只不过这次我们对前面的用户相似度矩阵做了一个转置。我们是不是可以直接利用原来的相似性测度函数啊？

Mr. 王：是的，前面的那种加权平均的形式，我们是可以直接利用的，其中的量稍作修改即可：

$$r_{xi}=\frac{\sum_{j\in N(i;x)}S_{ij}\cdot r_{xj}}{\sum_{j\in N(i;x)}\cdot S_{ij}}$$

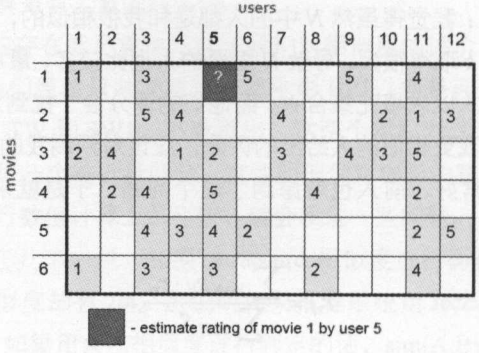
此时式中，S_{ij} 表示的是 i 和 j 项目的相似度。j 就是来自 x 评价过的与 i 最相近的项目。r_{xi} 是 i 的效用矩阵。

我们来看一个例子。有一个电影网站，近期比较热门的 6 部电影分别对应表格的 6 行，12 个进行过评价的用户就对应表格的 12 列。在这个例子中，评分的范围是 1 ~ 5。表格中灰色部分是已经进行过评分的数据域，白色部分是没有进行过评分的数据域。



	1	2	3	4	5	6	7	8	9	10	11	12
1	1		3			5			5		4	
2			5	4			4			2	1	3
3	2	4		1	2		3		4	3	5	
4		2	4		5			4			2	
5			4	3	4	2					2	5
6	1		3		3			2			4	

比如我们现在要对用户 5 对电影 1 的评分进行一个预测估计，也就是表格中标着问号的深灰色部分。



	1	2	3	4	5	6	7	8	9	10	11	12
1	1		3		?	5			5		4	
2			5	4			4			2	1	3
3	2	4		1	2		3		4	3	5	
4		2	4		5			4			2	
5			4	3	4	2					2	5
6	1		3		3			2			4	

现在我们总结一下协同过滤的优缺点。它的第一个优势就是适用于任何形式的项目，它完全不依赖项目的内容，不需要进行特征选择，直接通过用户的评分来进行衡量。不过它也存在非常明显的冷启动问题，协同过滤需要系统中有足够的用户进行匹配，无法推荐一个没有被评价过的项目，比如新项目和隐秘项目。我们得到的评分效用矩阵往往也是非常稀疏的，大量的用户都没有评价过相同的项目，这直接导致了我们很难去发现评价过相同项目的用户。另外，协同过滤无法给只有单一口味的用户推荐项目，它推荐的内容会比较分散，不像是基于内容的推荐那么集中，倾向于推荐流行项目。

我们发现，基于内容的推荐和协同过滤都各有优缺点，而且它们的优势和缺陷是互补的，我们可以尝试去实现两种或多种不同的推荐方法相结合的方式，并组合预测结果。比如可以结合基于内容的推荐方法与协同过滤，将我们研究过的两种方法进行组合，充分发挥它们的优势，避免它们的劣势。

小可：嗯，这两种方法如果单独使用确实显得极端了一点，一种只考虑了项目本身的特征；另一种只考虑了他人的评价，如果把它们结合起来确实更能发挥二者的优势，给出更加准确的推荐结果。

Mr. 王：别忘了，我们设计任何一种算法或者方法时，都要对其进行评价。这也是非常重要的。在不同的场合下，不同的方法给出的推荐准确程度是不一样的，我们依然要建立一种评价机制，对推荐模型的好坏进行评估。

第一个要考虑的就是准确度。对系统性能评价最重要的一部分就是推荐的内容是不是准确、恰当。

小可：想要推荐的结果足够准确，从前面两种方法来看，就需要我们的评分足够准确。评分的准确是推荐准确的基础。

Mr. 王：没错，对于这种基于评分的推荐模式，我们对项目的评分估计的准确程度还是非常重要的。常用的评分评价方法就是对比预测值与已知的评分。我们可以让评分预测模型去预测一些已知分数的值，然后用一些常用的数学误差评价机制，比如方均根误差、TOP 10 精度以及秩相关等方法对其进行评价。

另外，从预测的结果来看，我们还要从信息检索的角度对其进行评价，比如覆盖度，也就是系统能够预测的 item/user 数量，还可以让用户进行精确度评价，看看用户对系统的推荐是不是足够满意。另外，从受试者工作特征的角度，我们还可以对虚报率与漏报率进行测量。虚报就是那些被推荐而用户并不喜欢的项目，漏报就是那些用户本身会喜欢，却没推荐的项目。这两者的均衡曲线是可以用来衡量推荐系统效果的。

小可：均衡曲线？这说明它们往往不能同时被降低吧。

Mr. 王：一般来说是这样的，如果降低虚报率，往往就要尝试返回比较少的结果，这样会导致增大漏报率；如果降低漏报率，往往就要多返回一些结果，这就会导致虚报率增大。这两者往往需要进行平衡，很难同时达到最优。

另外，虽然我们需要去关注精度，但是有时狭隘地关注精度没有意义，推荐系统运用的环境往往是多样而复杂的，可能在我们的测试数据环境下表现得很好的推荐系统，在实际应用中效果就差了很多，即使在不同的情境下，不同的心情和不同的时间用户的喜好都有可能产生不同的变化。即使再好的模型，也会受制于很多很复杂的数据因素和人为因素，所以精度很多时候可以作为实验时的一个参考，不能太过迷信精度的作用。

小可：我想除了结果的好坏，得出结果的速度也是要考虑的吧。

Mr. 王：是的，我们要进行推荐的数据集合往往是非常复杂而庞大的。就以协同过滤为例，我们要操作的数据集合是一个非常庞大的矩阵，这是非常费时而困难的。其中最费时的步骤是找到 k 个最相似的用户，这一步骤的时间复杂度为 $O(|X|)$。在实际的推荐系统中，我们不能实时完成这个工作，如果等用户需要推荐了才开始考虑的话，那么是很难在用户可以接受的时间范围内完成的，反映在用户那里就是延迟非常大。为了避免这种情况的发生，我们可以进行预先计算。从以往的预先计算来看，朴素预先计算的时间复杂度为 $O(N \cdot |C|)$。

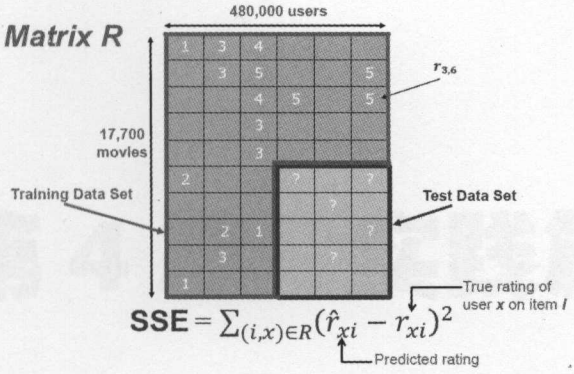
除了预先计算之外，对于如此之大的数据规模，我们也要尝试使用一些大数据处理方法，比如高维数据中的最近邻居搜索（LSH）、聚类、降维等思想，来简化和降低实际操作的数据量，以达到提升系统性能的效果。

Mr. 王：前面我们介绍了几种比较经典的推荐系统的策略，而目前在实际使用的各种推荐系统中，在推荐过程中考虑的因素还是非常多的。历史上曾经举办过一个推荐系统大赛，叫作 Netflix Prize。进行比赛的环境就是我们在协同过滤模型处理的那种环境，现有一些观影者对一些电影评分的矩阵，参赛者要用自己的模型进行处理，去预测矩阵中的缺失值。当年这场比赛的训练集包括 100 000 000 次评分、480 000 个观影用户和 17 770 部电影，横贯从 2000 年到 2005 年。要求是通过评估填充整个矩阵中缺失的 280 000 000 个评价。

小可：真是好大的数据集合，处理起这些数据来一定很困难。但当时的比赛用什么标准来衡量各支队伍的结果好坏呢？

Mr. 王：比赛采用了经典的均方根误差 RMSE。Netflix 自己的推荐引擎 Cinematch 能够达到的方均根误差为 0.9514。在当年参与比赛的超过 2700 支的队伍中，Netflix 给那些能在 Cinematch 引擎的结果上增强 10% 的队伍 100 万美元的奖金。

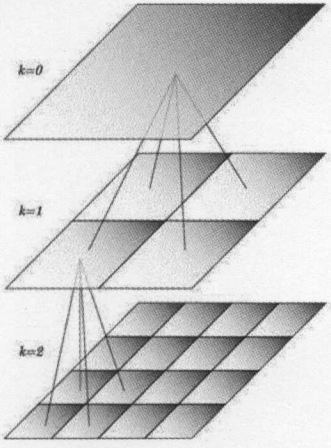
将这种问题变成一种模型，就是这样：



在这个图中，深灰色区域是训练集，其中包括用户对电影评分的已知数据；图中黑粗线框包围的区域是测试集，其对于参赛者来说是未知的。比赛的评价规则非常简单，就是看看参赛系统给出的结果和实际值的误差如何。

小可：那么最后谁获胜了呢？

Mr. 王：Netflix Prize 大赛的获胜者是 BellKor 推荐系统。它利用的方法叫作潜在因素模型，对数据进行了多尺度建模。它不仅考虑了我们前面讨论的全局特征，还研究了用户 / 电影的总体偏差、区域特征、局域特征，在这些特征的影响之下，基于协同过滤模型进行评分估计，最终获得冠军。关于潜在因素模型的内容比较复杂，我们就不在这里讨论了。



Mr. 王：这场比赛的受关注程度和奖金足见计算机科学界对推荐系统研究的重视，大量科学家和工程师们投身于推荐系统的研发和改进之中，他们留下了许多很有价值的成果和文献，课后你可以去翻阅相关资料，看看各大网站都使用了哪种推荐算法和策略。

第4篇　实践篇

第11章 磁盘算法实践

Mr. 王：前面讨论了很多理论方面的内容，从今天开始，我们研究如何从实践的角度去进行磁盘算法、并行算法和众包算法的设计。

小可：嗯，我也很想实际写几个程序去操作前面提过的算法。

Mr. 王：那么我们就从磁盘算法的实践开始吧。

小可：我们平时使用的计算机上的数据很多都是以文件形式进行存储的，那么是不是只要借助 C 语言读写文件的函数就可以操作磁盘了呢?

Mr. 王：文件的确是存储在磁盘上的，读写文件的操作也的确会产生磁盘读写。不过这样做大量的操作都是操作系统帮助我们完成的，对磁盘读写的大量细节我们并没有看到，在这里我会通过一些基本的程序，展示一个磁盘算法读写磁盘时的很多细节操作。

小可：嗯，在很多底层的操作中，操作系统和高级语言中封装好的函数为我们完成了太多的工作。

Mr. 王：现在我们就来深度剖析读写磁盘的过程。

首先给出两个用 C/C++ 语言读写磁盘的程序。

```
//read bytes in buf with size length
//the bytes is read to page
//the position is (blockid, offset)
void readBytes(BufMgr* bm, char *buf, int length, char *& page, int& blockid,
int& offset)
{
    if(length< bm->getPagesize() - offset){
            memcpy(buf,page + offset,length);
            offset += length;
```

```
        }
        else{
                int temp = bm->getPagesize() - offset;
                memcpy(buf, page+offset, temp);

                bm->UnpinPage(blockid);
                blockid += 1;
                bm->PinPage(blockid, page);

                memcpy(buf+temp, page, length - temp);
                offset = length- temp;
        }
    }

    //write bytes in context with size length
    //the bytes is write to page
    //the position is (blockid, offset)
    void writeBytes(BufMgr* bm, char *&page, char *context, int length, int&
page_no, int& offset)
    {
        if(bm->getPagesize()-offset >= length){
                memcpy(page+offset,context,length);
                offset += length;
        }
        else{
                int temp = bm->getPagesize() - offset;
                memcpy(page+offset, context, temp);
                bm->UnpinPage(page_no,true);
                page_no += 1;
                offset = 0;
                bm->PinPage(page_no, page);
                memcpy(page, context+temp, length - temp);
                offset = length-temp;
        }
    }
```

Mr. 王：上面两个函数分别是从磁盘读取字节数据和向磁盘写入字节数据的实现代码。我们来研究一下这段代码，暂时不关心其中一些函数的实现细节，先来看看这两个函数完成了什么工作。

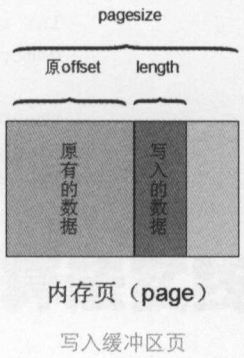
先说说写磁盘程序。

首先要明确的一点是，并不是每当要进行磁盘读写时，都直接读写磁盘，这样做是非常不经济的。所以当要读写磁盘时，就需要在内存中开辟一块空间，称作 Buffer（缓冲区）。

在写磁盘程序的开始，我们先检测缓冲区的页上还有没有剩余的空间去容纳所要写入的内

容。如果有，就直接把它们先存放在缓冲区中，并且将缓冲区的偏移量位置向前移动。

```
if(bm->getPagesize()-offset >= length)
{
    memcpy(page+offset,context,length);
    offset += length;
}
```



但如果页上剩余的空间不足以容纳所要写入内容的大小，那么就先将这个内存页剩余的部分填满。

```
int temp = bm->getPagesize() - offset;
memcpy(page+offset, context, temp);
```

接下来，对当前操作的内存页执行 Unpin 操作。

```
bm->UnpinPage(page_no,true);
```

然后增加页的编号，并且将偏移量归零。

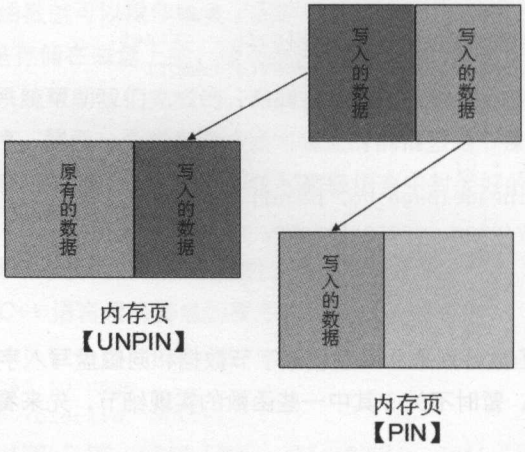
```
page_no += 1;
offset = 0;
```

再对新到达的内存页执行 Pin 操作。

```
bm->PinPage(page_no, page);
```

将在前一个页上还没有装满的数据填满，并设置偏移量。

```
        memcpy(page, context+temp, length - temp);
        offset = length-temp;
    }
}
```



小可：等等，老师，其中涉及了两个概念 Pin 和 Unpin，这是什么意思呢？

Mr. 王：这是磁盘操作中十分关键的两个操作。之前我们也讨论过，在操作磁盘的过程中，我们并不会直接去操作磁盘，而是将磁盘块加载到内存中来，在内存中进行操作和处理。读写磁盘也是一样，我们会在内存中建立一个缓冲区，在缓冲区中存放要操作的磁盘块的数据镜像。

此时当需要读取某一部分数据时，如果磁盘中已经存在这部分数据，就不必要再从磁盘中读取数据了，而是从内存中读取该数据。

小可：嗯，毕竟操作内存要比操作磁盘快得多。

Mr. 王：可是，由于我们在不停地对磁盘进行着读写操作，所以很多时候，缓冲区中的数据和磁盘中的数据并不是一致的，这就会带来很严重的问题——“脏数据”。

下面举一个例子来说明这个问题，

比如有一个进程正在修改一个磁盘块的内容，而另一个进程要读取该内容，此时磁盘块中的数据并没有修改完成，所读取的数据并不是正确的数据。

为了避免发生这种读取不一致数据的问题，我们引入了基本操作——Pin 和 Unpin。

正像它们的名字一样，Pin 是用一个“针”把这个磁盘块给“钉住”，不允许其他进程来读取它的值，因为有一个进程正在写它，这时候读取的值是不正确的。Unpin 正相反，就是写这个磁盘块的过程已经结束，现在内存中的缓冲区和磁盘块中的内容已经一致，可以安全地读取其数据了，此时就不必再“钉住”它了。

小可：哦，我懂了，简单来说，就是防止其他进程在写的过程中读取了正在被写的数据。

Mr. 王：是的。类似的机制在很多两级存储器中有所体现，比如内存和高速缓存（Cache）之间也有相似的机制，以防止 CPU 读到 Cache 和内存中不一致的数据。

小可：可是 Pin 和 Unpin 又是如何实现的呢？

Mr. 王：接下来我们就来谈谈 Pin 和 Unpin 的实现。

其实 Pin 和 Unpin 这两个操作的原理很简单，我们只需要维护一个查找表，这个查找表标记着各个磁盘块和其对应的内存缓冲区的状态。

PinPage 函数完成了这样的功能：对 id 为 pid 的页执行 Pin 操作，程序会将磁盘中非空且不在缓冲区中的对应页加载到内存中。

程序在执行过程中，首先会在缓冲区中为新来的页寻找空间，如果没有足够的空间，程序会从缓冲区中寻找一个页替换出去，以提供空间给新的页；如果仍然找不到这样的页，就会报错。如果读入的页不是空的，则将正常执行 Pin 操作，并且在 Hash 表中存储页号和帧号，以标记这是一个已加入缓冲区的被 Pin 页。

下面是 PinPage 的源代码。

```
Status BufMgr::PinPage(PageID pid, Page*& page, BOOL isEmpty)
{
    int frameNo;
    m_nTotalPin++;
    frameNo = FindFrame(pid);
    if (frameNo == INVALID_FRAME)
    {
        BOOL bWriteBack;
```

```
            frameNo = replacer->PickVictim(bWriteBack);
            if( bWriteBack ) { m_nTotalWrite++; }
            if (frameNo == INVALID_FRAME)
            {
                printf("ERROR: Buffer pool is full");
                exit(1);
                return 2;
            }
            if (!isEmpty)
            {
                frames[frameNo]->Read(pid);
                m_nTotalMiss++;
                if( m_LastMissPid + 1 != pid )
                {
                    m_nTotalRandMiss++;
                }
                m_LastMissPid = pid;
            }
            else
            {
                frames[frameNo]->SetPageID(pid);
            }
            hashTable->Insert(pid, frameNo);
        }
        frames[frameNo]->Pin();
        page = frames[frameNo]->GetPage();
        return 1;
    }
```

小可：等等，老师，这里提到了一个概念 Hash 表，这是什么呢？

Mr. 王：嗯，这是一个应用非常广泛的数据结构，跟你讲讲它的原理吧。Hash 表又叫散列表，是一种非常常见的用于实现数据字典的数据结构。它的原理非常简单，却能实现非常高效的插入、删除和查找。其时间复杂度为 O(1)。

小可：这么快，常数时间的查找在以前提到过的数据结构中还是非常少见的啊！

Mr. 王：先来谈谈散列表的原理。其之所以能够以这么快的速度进行查找，就是因为在散列表中，数据记录值和其所保存的位置（地址）之间有着非常强的直接关联。一般来说，最常见的散列表的空间大小为一个素数 m，这个素数的大小根据要存储的数据多少来定。

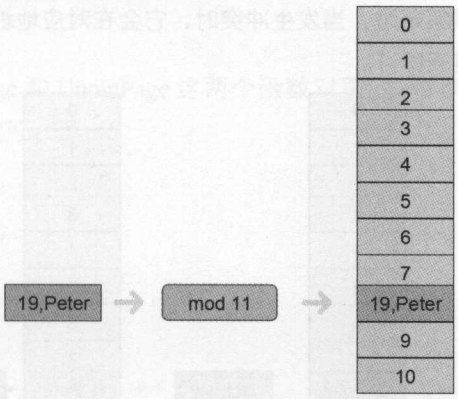
这里先以存储的关键字是数字为例。比如要存储学生的编号和姓名，一条记录就是 <num,name> 这样的结构，例如 <19,Peter>。假如散列表大小为 11。

每当要执行一个添加或者查找操作时，我们就进行如下运算：

$$address = key \bmod m$$

对于这个例子，address = 19 mod 11 = 8。

所以我们将 <19,Peter> 这条记录存储在散列表中的 8 号位置。



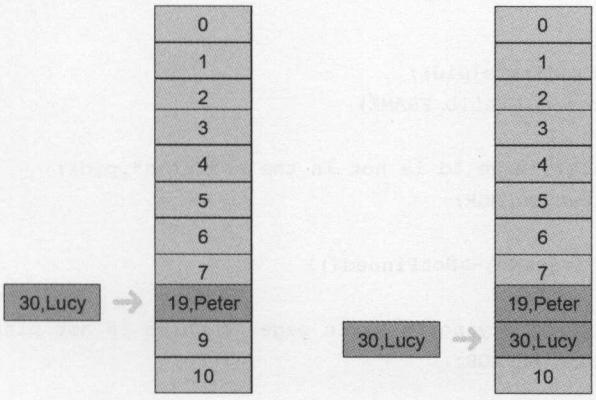
小可：哦！我明白了。同理，当我需要查找编号为 19 的记录时，需要做的同样是用这个编号去对 m=11 求余数，求出的结果 8 就是它的地址了。

Mr. 王：非常好，如果散列表是用数组实现的，每当我们拿到一个数据时，就可以直接求出它的数组下标，根据对应的下标查找数据就可以了。不难看出，查找一个数据和数据规模 n 没有关系，只需要通过对数据的值进行计算即可，所以时间复杂度是 O(1)。

小可：可是，19 mod 11 = 8，30 mod 11 也是 8，如果这个散列表中同时存在编号为 19 和 30 两条记录怎么办呢？

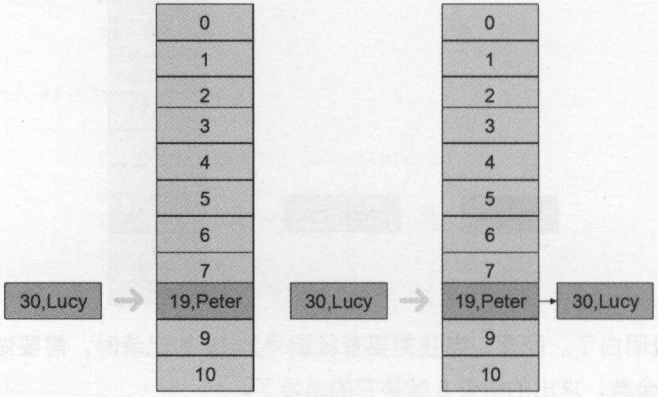
Mr. 王：很好，你注意到了这个问题。当数据逐渐增多时，很多时候两个数据要放在同一个地址中，散列表的内部会发生冲突。这时候我们就需要对这些冲突的数据进行处理，常见的方法有内散列和外散列。

内散列的策略是，当发生冲突时，新添加的数据放置到下一个位置上，这时只需要增加一个判断，判断该地址上是不是已经有数据了即可。



内散列的缺点是很明显的，一旦发生了冲突，就会去冲撞其他的地址，引起的冲突会越来越严重。

外散列又叫拉链法，其策略是，当发生冲突时，它会在对应地址的外部创建一个链表，当查找数据时，去查找这个链表即可。



相比之下，外散列方法会更加常用一些。

Mr. 王：当散列表中的数据不那么多，冲突不严重时，可以近似达到 O(1)；而当其中存储的数据较多时，性能会发生一定程度的下降，但散列表仍然不失为一种非常好的查找结构。关于散列表的寻址方法，这里只是举了一个最简单、最经典的散列函数例子，散列函数除了取模这种方法外，还有很多方法，如果你感兴趣，可以去查阅关于散列表的资料。

相应的，Unpin 操作是将编号为 pid 的页解除 Pin 状态。如果这个页是一个脏页，则要对其进行标注。所谓的脏页，就是磁盘和内存缓冲区中的内容不一致的数据。其执行的前置条件是，这个页本身一定是保存在缓冲区中，并且是已经被 Pin 过的，否则会向用户报错。

下面是 UnpinPage 的源代码。

```
Status BufMgr::UnpinPage(PageID pid, BOOL dirty)
{
    int frameNo;
    frameNo = FindFrame(pid);
    if (frameNo == INVALID_FRAME)
    {
            printf("Page %d is not in the buffer\n",pid);
            return BUFMGR;
    }
    if (frames[frameNo]->NotPinned())
    {
            printf(" Trying to unpin page %d which is not pinned\n", pid);
            return BUFMGR;
    }
```

```
    if (dirty)
            frames[frameNo]->DirtyIt();
    frames[frameNo]->Unpin();
    return OK;
}
```

Mr. 王：其中，PinPage 和 UnpinPage 这两个函数对页的 Pin 和 Unpin 操作，只需要修改 Pin 和 Unpin 帧数的计数就可以了。

```
void Frame::Pin()
{
    if( m_pinCount == 0 )
    {
            m_nUnPinnedFrames--;
    }
    m_pinCount++;
}

void Frame::Unpin()
{
    m_pinCount--;
    if( m_pinCount == 0 )
    {
            m_nUnPinnedFrames++;
    }
}
```

第12章 并行算法实践

12.1 Hadoop MapReduce 实践

12.1.1 环境搭建

Mr. 王：前面我们讲了很多关于并行算法的理论，今天我们来看看如何在计算机上实际运行一些并行算法。

小可：我早就迫不及待想试试了。

Mr. 王：我们要先安装和配置 Hadoop。

前面我们提到过，Hadoop 是 MapReduce 的一个开源实现版本，如今的 Hadoop 已经成为了包含许多部分的独立集合，比如 Hive、HBase、ZooKeeper 等。但从根本上讲，Hadoop 的基本组成部分主要有两个：一个是 MapReduce；另一个是 HDFS。

小可：MapReduce 我知道，是并行计算的编程框架，那 HDFS 是什么呢？

Mr. 王：HDFS 是 Hadoop 分布式文件系统。由于 Hadoop 是一个并行计算的框架，这意味着它保存的文件往往是存储在多台计算机上的，所以 Hadoop 也必须能够管理多台计算机上文件的机制，这就是 HDFS。

一个配置了 HDFS 的机群包括一个 NameNode 和数个 DataNode。其中 NameNode 就像 MapReduce 中的 Master，负责管理整个文件系统中文件的命名和用户对文件的访问操作等；而 DataNode 相当于 Slave，负责存储具体的文件和数据。

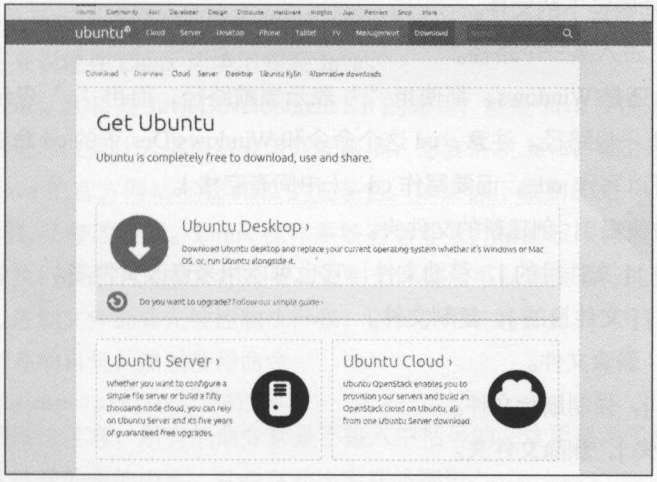
小可：嗯，那么 Hadoop 是如何安装和使用的呢？

Mr. 王：一般来说，Hadoop 平台都安装在 Linux 操作系统下，当然在 Windows 操作系统下也是有办法使用它的，不过一般需要一个 Linux 终端模拟器，比如 Msys 或者 Cygwin 等。不过原生的 Hadoop 是运行在 Linux 环境下的，这里我建议使用 Linux 操作系统来安装和配置 Hadoop。

Mr. 王：Linux 操作系统最初是由程序员 Linus Torvalds 开发的，是一个类 UNIX 内核的支持多用户、多线程、多任务、多处理器的操作系统。Linux 是一款广受开发人员喜爱的开源操作系统，有很多组织和社区为它服务。现在市面上有大量的 Linux 操作系统的发行版本，这些基于 Linux 内核的操作系统统称为 Linux。

小可：我听说过，很多初学者都使用 Linux 非常著名的发行版本之一 Ubuntu Linux。

Mr. 王：是的，配置和安装 Linux 的桌面版本非常容易，现在网络上有很多 Linux 的安装资源，可以从其官网上下载安装 Ubuntu Linux 并将其烧录到一个启动 U 盘里面，以便进行 Hadoop 的实践。



小可：嗯，我的计算机中已经安装好了 Linux。

Mr. 王：在安装好 Linux 之后，我们还需要先配置 Java 环境。因为 Hadoop 的原生开发语言是 Java，之后我们进行的所有开发都是使用 Java 语言。当然，Hadoop 也为喜欢使用 C++ 语言的用户开发了 C++ 编程接口。不过，在这里我们使用 Java。

虽说目前很多 Linux 发行版本都有很漂亮的图形界面，但是大多数开发者在 Linux 下用来操作计算机的环境都是终端（Terminal），终端类似于 Windows 下的命令提示符，是一种通过命令行来操作计算机的方式，如果对操作系统的命令比较熟悉的话，操作是非常方便和快捷的。Ubuntu Linux 的终端承载的是 Bash Shell。

小可：不太懂，什么是 Shell 呢？

Mr. 王：在 Linux 操作系统中，我们将操作系统的内核称为 Kernel，是操作系统内部最基本的组成部分。而与用户进行交互的部分叫作 Shell，可以用来执行命令、程序、和脚本。这里不对其太过深究，如果你对 Linux 系统感兴趣的话，网络上有很多相关内容可以查阅。

这里给出一些常用的 Linux 命令，以便你能更加适应后面的命令操作。

我们在 Ubuntu Linux 的桌面下可以使用“Ctrl+Alt+T”快捷键来打开终端，或者使用左边的 Dash 启动器打开终端。

小可：和 Windows 的命令提示符很像，是输入命令的黑框框。

Mr. 王：现在我们看到屏幕上出现的是

[用户名]@[计算机名]~$_

前面部分显示的是用户的名字和本台计算机的名字，后面的小波浪线部分显示的应该是当前路径。而 Linux 默认的路径是 /home/[用户名] 文件夹，它也称作当前用户的主文件夹，在 Linux 中主文件夹常用“~”来代替。$ 符号是输入命令的界线，后面的就是命令了。

常用的命令如下。

- ls，显示当前路径下的文件。
- cd [路径]，进入某一个路径。

不论是 Linux 还是 Windows，都使用“.”表示当前路径，而用“..”表示上一级路径。所以 cd .. 就是返回上一级路径。注意，cd 这个命令和 Windows/Dos 下的 cd 命令略有不同，返回上一级文件夹不可以写作 cd..，而要写作 cd ..（中间有空格）。

- mkdir [文件夹名]，创建新的文件夹。
- mv [文件源] [文件目的]，移动文件，它也可以用来修改文件名。
- cp [文件源] [文件目的]，复制文件。
- rm [文件]，删除文件。
- rm -f [文件]，强制删除文件。
- rm -r [文件夹]，删除文件夹。
- rm -rf [文件夹]，强制删除文件夹。
- cat [文件]，查看文件的内容。
- ./[.sh]，执行当前路径下的脚本。注意不要遗漏“./”，否则会找不到脚本的位置。

小可边听边尝试着使用前面的命令。

Mr. 王：如果熟练的话，操作 Linux 系统的速度可以非常快。更重要的是，命令的操作可以在 Linux 下写成脚本。

可以打开一个文本文件：gedit 1.sh。脚本文件是以 .sh 为扩展名的。

打开后，首先写下固定的一条命令：

```
#!/bin/bash
```

然后只要将命令逐条地写在这个脚本文件中就可以了。比如：

```
cd ~/docdir
mkdir note
cd note
```

这样的一系列命令就可以通过 1.sh 的运行自动执行了。

我们可以使用 ./1.sh 命令来执行这个脚本。后面我们会看到很多的 .sh，这都是 Linux 的 Shell 脚本，我们可以通过书写一些简单的脚本对它们有一个初步的认识。其实在脚本中可以包含很复杂的逻辑，包括判断、循环、表达式匹配等。当有需要时，你可以去专门学习 Shell 脚本语言。

Linux 拥有非常有代表性的用户权限机制。当你在执行一些命令而提示你没有权限时，可以在命令的前面加上 sudo，这条命令可以使你以管理员身份执行命令。比如：

```
sudo shutdown -r 0
```

这条命令就是重新启动计算机，需要管理员权限，使用 sudo 命令来让自己具有管理员权限。

小可：提示我输入密码了！

Mr. 王：密码是最能证明你的管理员身份的验证方式。

好了，我们再来说说在 Linux 环境下安装 Java。

首先要下载 JDK。JDK 是 Java Development Kit 的缩写，就是 Java 开发工具包，是 Java 开发必备的环境。不论你使用什么编辑器或者开发环境，想要开发 Java，都必须安装 JDK。

首先我们从 Java 的官方网站下载 JDK 的安装包。如果你下载到的是一个 .bin 文件，则可以直接运行它，别忘了修改权限。Linux 操作系统有着非常严格的权限机制。在 Linux 操作系统看来，任何有着执行权限的文本文件或者二进制文件都是可以直接运行的，所以很多可执行的文本文件或者二进制文件都要先被控制执行权，以保护系统的安全。

现在进入它所在的目录，然后使用命令：

```
[Java 路径 ] $ sudo chmod u+x [.bin 文件的文件名 ]
```

来修改它的执行权限（注意，所有的命令都是不输入中括号的，这里是为了标注一些变量的存在，而且不输入 $ 符号前面的内容，其会自动出现在终端里）。

然后只要使用 sudo ./[.bin 文件的文件名] 命令即可。

另外，如果你下载到的 JDK 不是一个 .bin 文件，则可以将其解压缩到一个方便使用的目录下即可。

小可：这样是不是就安装完成了？

Mr. 王：虽然如此，但是想要使用 JDK，还是需要配置环境变量的。

小可：为什么要设置环境变量？

Mr. 王：这里设置环境变量是由于在终端中使用 Java 时，我们要事先告诉系统到哪里去找 Java，否则计算机就会找不到它。

我们在用户目录下打开 Bash Shell 的配置文件 .bashrc，其中 bashrc 前面的 "." 表示它是一个隐藏文件，在图形用户界面下一般是看不到它的，在输入文件名时，也注意不要丢掉它。

```
~$ gedit .bashrc
```

当然，如果你是 vim 用户，就使用：

```
~$ vim .bashrc
```

小可：vim 是什么呢?

Mr. 王：vim 是 Linux 下开源的文本编辑器，它的功能非常强大，受到广大编程爱好者的欢迎，非常适合用来编写程序代码等，它提供了自动的代码高亮功能。如果你经常在 Linux 下写程序的话，则可以尝试学习文本编辑器的使用，在文本编辑器中最著名的是 vim 和 emacs，它们都是非常不错的文本编辑器。

好了，打开 .bashrc 之后，可以对其进行修改，注意不要破坏其他部分。

在文件的尾部添加：

```
#My Java Environment
export JAVA_HOME = [ 安装路径 ]/[JDK 文件夹名 ]
export CLASSPATH = ".:JAVA_HOME/lib:$CLASSPATH"
export PATH = "$JAVA_HOME/bin:$PATH"
```

其中，# 号后面的内容都是注释，是为了方便我们以后找到它。

后面的三行方便系统找到 Java 运行环境、开发环境和工程，其中包含的冒号是分隔符，注意不要丢掉 CLASSPATH 的第一个 "."，否则容易出现找不到类的错误。

接下来输入：

```
java -version
```

终端会返回类似下面的信息：

```
java version "1.8.0_20"
Java(TM) SE Runtime Environment (build 1.8.0_20-b26)
Java HotSpot(TM) 64-Bit Server VM (build 25.20-b23, mixed mode)
```

这是 Java 的版本信息。如果终端能够正常地输出版本信息的话，则说明 Java 运行环境已经可以正常使用了。

然后输入：

```
javac -version
```

终端会返回 javac 的版本信息。如果终端能够正常地输出版本信息的话，则说明 Java 的编译器已经可以正常使用了。

网络上关于 Java 的安装教程是非常多的，如果在安装中和环境变量配置中出现问题，在网络上的一些博客和论坛中可以找到答案。

小可在自己的计算机上敲了一会儿，看着屏幕上输出的版本信息，说：嗯，我都已经配置好了，现在就可以安装 Hadoop 了吧。

Mr. 王：别急，我们还要安装一个 SSH。SSH 即安全外壳协议的缩写，是为了远程登录和

网络服务的一个安全通信协议。

小可：这是因为 Hadoop MapReduce 是提供多台计算机并行计算的平台，所以需要一个网络通信安全协议吧?

Mr. 王：是的。SSH 的安装并不是必要的，但是一般使用它来进行网络连接服务的安全代理；否则，当操作一批计算机时，每次进行连接都要输入目标机器的密码，这样使用起来非常不方便，我们可以用 SSH 协议来避免这个麻烦。

下面是安装 SSH 的命令，前面的 apt-get install 也是在 Ubuntu 下安装很多软件的方法。

```
~$ sudo apt-get install ssh
```

安装好之后，我们用 SSH 来建立一个公密钥对，公密钥对相当于一组钥匙和锁头的关系。

```
~$ ssh-keygen -t dsa -P "-f [ 保存公密钥对的路径和文件名 ]
```

为了方便寻找，建议路径为 ~/.ssh/id_dsa。

然后将这个公钥放到授权的公钥文件中。

我们复制 ~/.ssh/id_dsa.pub 里面的全部内容，将其放入 authorized_keys 中。

```
$ cat ~/.ssh/id_dsa.pub >> ~/.ssh/authorized_keys
```

现在执行命令：

```
ssh localhost
```

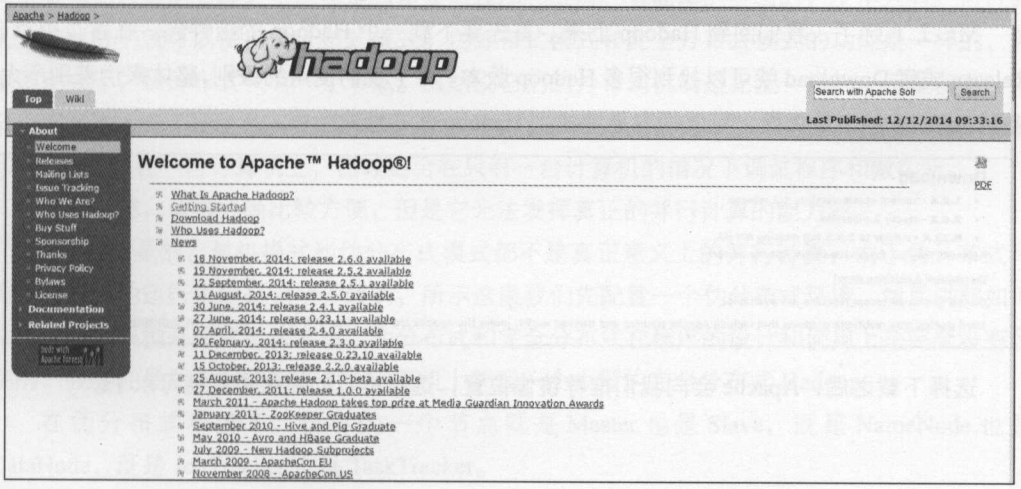
小可：返回了一些登录信息。

Mr. 王：这说明我们已经可以用 SSH 协议免密码登录到本机了。当我们用 Hadoop 平台操作大量的计算机时，一般要将这些计算机设为免密码登录。

小可：现在是不是可以安装 Hadoop 了?

Mr. 王：现在下载 Hadoop 的压缩包，其名字一般是 Hadoop 和它的版本号，比如 Hadoop-1.0.1.tar.gz，我们将其放在用户目录下，然后将其解压缩成一个文件夹。

首先我们登录 Hadoop 的官方网站：https://hadoop.apach.org。



Hadoop 的标识是一个非常可爱的小象加上蓝色的 Hadoop 字样。在这里我们可以找到 Hadoop 的很多发行版本和资料。

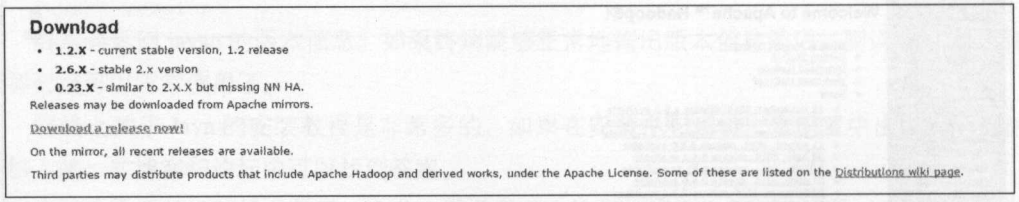
目前 Hadoop 包含以下几个基本的组成部分。

- Hadoop Common：这是 Hadoop 运行的依赖基础，是一些用于支持 Hadoop 运行的各种底层模块。
- HDFS：这就是我们前面提到的 Hadoop 分布式文件系统。为了能让文件分散存储在多台计算机组成的机群上，我们需要一种机制使得所有计算机的磁盘可以有机地结合成一个可以存放大量文件的文件系统，在 Hadoop 中这个文件系统就是 HDFS。
- YARN：带有 YARN 的 Hadoop 一般也被称作新一代的 Hadoop，或者 YARN。其实 YARN 是一个机群资源管理系统和任务表编排的框架，它的出现使得 Hadoop 的运行效率和稳定性得到了很大的提升。
- MapReduce：这就是我们熟悉的 MapReduce，也就是使用 Hadoop 进行并行数据处理的核心框架。

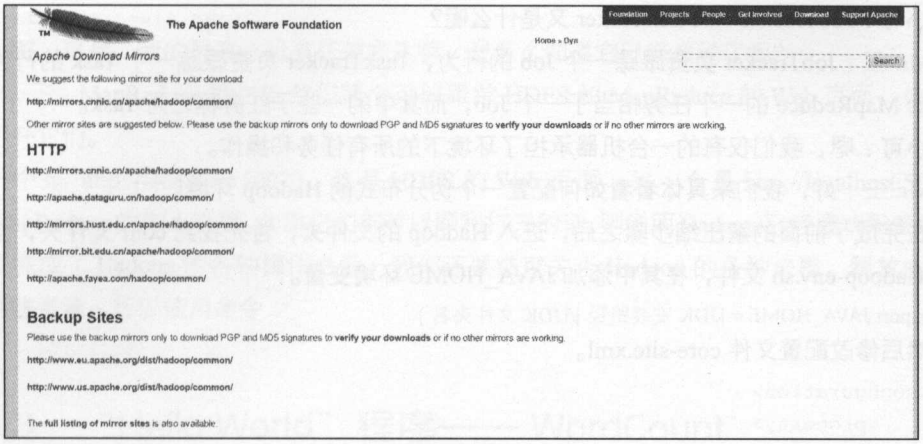
另外，Hadoop 也提供了大量的适用于各种不同任务的组件包和工具包。

- Hive：基于 Hadoop 的数据仓库工具。
- Mahout：一个用于机器学习、数据挖掘的库。
- HBase：分布式数据库系统。
- Chukwa：用于监控大型分布式系统的数据收集系统。
- Pig：为用户提供多种接口的大数据分析平台。
- ZooKeeper：一个分布式的应用程序协调服务。
- 还有我们在后面要讲到的 Spark。

Mr. 王：好了，我们回到 Hadoop 上来。首先要下载一个 Hadoop 的组件包。在首页中点击 Release 或者 Download 就可以找到很多 Hadoop 版本。对于我们使用的级别，整体来讲差别不大。来到 Hadoop 的下载页面，可以很容易地找到 Hadoop 发布版本



选择下载之后，Apache 会向我们推荐镜像位置，选择推荐的镜像位置就可以了。



下载之后，将其解压缩到一个我们能够找到的目录就可以了。

小可：嗯，我已经解压缩好了！

12.1.2　配置 Hadoop

在开始使用 Hadoop 之前，先要对 Hadoop 进行配置。Hadoop 的配置分为单机模式、完全分布式、伪分布式三种。单机模式一般用于系统的调试，我们不去使用它。当我们要在机群上执行真正的大数据并行计算时，需要使用完全分布式模式才能让并行计算顺利完成。也只有在完全分布式模式下，才能真正地发挥并行计算的效果。

小可：那什么是伪分布式呢？

Mr. 王：我们知道，分布式系统是基于网络的多机计算系统。也就是说，至少要有两台计算机参与到任务的处理之中。但是当需要写程序和进行一些简单的实验时，手中只有一台计算机，这时我们就可以使用伪分布式模式。伪分布式模式和完全分布式模式的功能是一样的，但是区别在于，伪分布式仅有一个节点。虽然说它们的分布式机制是完全一样的，但是仅有一个节点在实质上又不能称作分布式系统，所以称为伪分布式。在伪分布式环境下，我们将 Master 和 Slave 都放在一台计算机上，比较适合在只有一台计算机的情况下调试程序和做实验。

小可：嗯，这样的确比较方便，但是它无法发挥真正的并行计算的能力吧。

Mr. 王：是的，单机模式和伪分布式模式都不是真正意义上的并行计算，由于伪分布式和完全分布式的运行机制是完全一致的，所示这里我们先配置一个伪分布式环境，然后讲解如何使用它。如果抛开配置不谈的话，伪分布式和完全分布式在程序的设计和使用上是完全没有区别的，我会在最后介绍如何在 3 台计算机上部署一个小型的完全分布式 Hadoop。

在伪分布式环境下，仅有的一个节点既是 Master 也是 Slave，既是 NameNode 也是 DataNode，既是 JobTracker 也是 TaskTracker。

小可：JobTracker 和 TaskTracker 又是什么呢？

Mr. 王：JobTracker 负责跟踪一个 Job 的行为，TaskTracker 负责跟踪一个 Task 的行为，我们交给 MapReduce 的一个任务相当于一个 Job，而其中的一些子任务称之为 Task。

小可：嗯，我们仅有的一台机器承担了环境下的所有任务和操作。

Mr. 王：好，我们来具体看看如何配置一个伪分布式的 Hadoop 环境。

在完成了前面的解压缩步骤之后，进入 Hadoop 的文件夹，首先找到 conf 文件夹，修改里面的 Hadoop-env.sh 文件，在其中添加 JAVA_HOME 环境变量。

```
export JAVA_HOME = [JDK 安装路径 ]/[JDK 文件夹名 ]
```

然后修改配置文件 core-site.xml。

```
<configuration>
    <property>
        <name>fs.defaultFS</name>
        <value>hdfs://localhost:9000</value>
    </property>
</configuration>
```

这一步我们编辑的是 Hadoop 的配置文件，在这里要指定 HDFS 的地址和端口号。

然后修改 MapReduce 的配置文件。

```
<configuration>
    <property>
        <name>mapred.job.tracker</name>
        <value>localhost:9001</value>
    </property>
</configuration>
```

这里我们对 MapReduce 的 JobTracker 的地址和端口号进行配置。

最后修改配置文件 hdfs-site.xml。

```
<configuration>
    <property>
        <name>dfs.replication</name>
        <value>1</value>
    </property>
</configuration>
```

这里我们修改的是 HDFS 的配置文件，要确定运行模式为伪分布式模式。

至此，伪分布式的 Hadoop 配置就全部完成了。不过在使用之前，我们还要对 HDFS 进行格式化。

小可：HDFS 还真像一块磁盘，在使用之前还要进行格式化。

Mr. 王：我们进入 Hadoop 的文件夹，然后执行命令：

```
$ bin/hdfs namenode -format
```

接下来就可以运行 MapReduce 试一下了，我们可以用自动化脚本直接启动所有的进程。

```
$ bin/start-all.sh
```

小可：可是，我的屏幕上没有任何变化啊，我怎么知道它已经启动了呢？

Mr. 王：MapReduce 提供给我们两个可以跟踪 HDFS 和 MapReduce 的 Web 页面，使用浏览器打开它们。

一个是 http://localhost:50070，这是 HDFS 的 Web 页面；另一个是 http://localhost:50030，这是 MapReduce 的 Web 页面。如果它们都可以顺利打开的话，则说明 Hadoop 已经成功配置好了。

在完成了 Hadoop 的各种操作之后，我们还要结束关于 Hadoop 的各种进程，释放由其占据的系统资源，可以使用命令：

```
$ bin/stop-all.sh
```

12.1.3　“Hello World”程序—— WordCount

Mr. 王：你知道“Hello World”吗？

小可：嗯，我在学习 C 语言时，学会的第一个程序就是向屏幕输出一行“Hello World!”。后来程序员们就喜欢用 Hello World 来代指学习每一种语言的第一个程序。

Mr. 王：今天我们就来学习 Hadoop 的“Hello World”。

小可：它不会也是向屏幕输出一行 Hello World 吧？

Mr. 王笑着说：当然不是了，这样简单的任务如何能够体现一个并行计算系统的效果呢？前面我们讲过关于 WordCount 的内容。WordCount 是 Hadoop 最基本的一种应用，有很多人说 WordCount 就是 Hadoop 的“Hello World”。

小可：嗯，WordCount 就是统计文章中单词出现的个数。那么应该怎么去实现它呢？

Mr. 王：我们先来看看代码是如何实现的。我们知道 Hadoop 的原生开发语言是 Java，所以这个程序也使用 Java 来编写。程序清单如下：

```
import java.io.IOException;
import java.util.*;

import org.apache.hadoop.fs.Path;
import org.apache.hadoop.conf.*;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapred.*;
import org.apache.hadoop.util.*;

public class WordCount
{

    public static class Map extends MapReduceBase implements Mapper<LongWritable,
Text,Text,IntWritable>
    {
```

```
        private final static IntWritable one =new IntWritable(1);
        private Text word= new Text();

        public void map(LongWritable key,Text value,OutputCollector<Text,
IntWritable> output,Reporter reporter) throws IOException
        {
                String line = value.toString();
                StringTokenizer tk = new StringTokenizer(line);
                while (tk.hasMoreTokens())
                {
                        word.set(tk.nextToken());
                        output.collect (word,one);
                }
        }
    }
    public static class Reduce extends MapReduceBase implements Reducer<Text,
IntWritable,Text,IntWritable>
    {
        public void reduce (Text key,Iterator<IntWritable> values,
OutputCollector<Text,IntWritable> output,Reporter reporter)throws IOException
        {
                int sum =0;
                while (values.hasNext())
            {
                        sum+= values.next().get();
                }
                output.collect(key,new IntWritable(sum));
        }
    }


    public static void main(String[] args) throws Exception
    {
        JobConf conf = new JobConf(WordCount.class);
        conf.setJobName("wordcount");

        conf.setOutputKeyClass(Text.class);
        conf.setOutputValueClass(IntWritable.class);

        conf.setMapperClass(Map.class);
        conf.setReducerClass(Reduce.class);

        conf.setInputFormat(TextInputFormat.class);
        conf.setOutputFormat(TextOutputFormat.class);
```

```
        FileInputFormat.setInputPaths(conf,new Path(args[0]));
        FileOutputFormat.setOutputPath(conf,new Path(args[1]));

        JobClient.runJob(conf);
    }
}
```

我们来分析一下这个程序。

程序的第一部分是包的引用，这里包括一些我们会用到的常见的 Java 包，也包括一些 Hadoop 中的包。

```
import java.io.IOException;
import java.util.*;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.conf.*;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapred.*;
import org.apache.hadoop.util.*;
```

第二部分，在类中定义内嵌类 Map，也就是 MapReduce 中的 Map 部分。首先，Map 是 MapReduceBase 基类的派生类，这个类定义在 Hadoop 的包中；其次，它也是对接口 Mapper 的实现，其中要对 Mapper 接口的模板类型进行指定。

```
public static class Map extends MapReduceBase implements
Mapper<LongWritable,Text,Text,IntWritable>
```

小可：我以前从来没见过 LongWritable、Text、IntWritable 这几种类型，它们有什么用呢？

Mr. 王：这几种类型是 Hadoop 中定义的，它们用于封装 Java 中的 long、int、string 等类型。Hadoop 对这些类型进行了重新设计和定义，使之更适合于分布式、并行的环境。读代码时，只要把它们看作 long string 和 int，写代码时，记得用它们替换 Java 中的基本类型就可以了。

```
public void map(LongWritable key,Text value,OutputCollector<Text,IntWritable>
output,Reporter reporter) throws IOException
```

Mr. 王：这是对 Map 类中 map 方法的定义，由它来完成 Map 的主要工作。在参数列表中，key 和 value 两个变量对应着 MapReduce 中的 key-value 对；而 OutputCollector<Text,IntWritable> 对 Mapper 的输出类型进行了定义，在 WordCount 的设计中，Mapper 要输出的就是 <word,1> 这样的键值对，所以我们定义的 OutputCollector 就应当匹配 word 的类型 Text 和 1 的类型 IntWritable。

小可：嗯，下面这部分就比较好理解了，使用一个 StringTokenizer 类将原文中连起来的长字符串内容切分成一个个单词，然后将单词逐个地和 1 组合并发送出去。这是符合我们对 Mapper 的定义的。

```
String line = value.toString();
StringTokenizer tk = new StringTokenizer(line);
```

```
while (tk.hasMoreTokens())
{
    word.set(tk.nextToken());
    output.collect (word,one);
}
```

Mr. 王：Reducer 和 Mapper 的定义是相类似的。我们也要定义内嵌类 Reduce，并且它也是 MapReduceBase 基类的派生类，这一次要实现的接口是 Reducer。

```
public static class Reduce extends MapReduceBase implements Reducer<Text,
IntWritable,Text,IntWritable>
{
    public void reduce (Text key,Iterator<IntWritable> values,OutputCollector<
Text,IntWritable> output,Reporter reporter)throws IOException
```

小可：嗯，下面这部分也是比较好理解的，Reducer 会对收集来的 <word,1> 进行统计，将相同的单词的值累加起来。

```
    int sum =0;
    while (values.hasNext())
    {
            sum+= values.next().get();
    }
    output.collect(key,new IntWritable(sum));
JobConf conf = new JobConf(WordCount.class);
conf.setJobName("wordcount");

conf.setOutputKeyClass(Text.class);
```

Mr. 王：其中主函数是对 Hadoop 的运行进行一些配置，顾名思义，可以通过直接阅读这些英文单词来了解它们的意义。比如 setOutputValueClass 就是设定输出值的类型，setMapperClass 就是设置 Mapper 的类型等。

```
conf.setOutputValueClass(IntWritable.class);
conf.setMapperClass(Map.class);
conf.setReducerClass(Reduce.class);
conf.setInputFormat(TextInputFormat.class);
conf.setOutputFormat(TextOutputFormat.class);
FileInputFormat.setInputPaths(conf,new Path(args[0]));
FileOutputFormat.setOutputPath(conf,new Path(args[1]));
```

我们将 MapReduce 执行的一个工作称作一个 Job，在完成了所有的定义和配置之后，我们要让 Job 运行起来，用的就是最后一句代码：

```
JobClient.runJob(conf);
```

小可：我觉得这个程序挺简单的，除了和 MapReduce 接口相关的一些内容以外，就像我们平时写的单机程序一样。

Mr. 王：这就是 MapReduce 平台的强大之处，内部处理的很多问题，比如网络的连接、通

信、具体的任务分配等都不需要开发者去规定和设计，只要基于 MapReduce 这个框架，就可以免去使用者的很多麻烦。

小可：的确，那我们来看看运行结果吧。直接运行这个 Java 程序就可以了吗?

Mr. 王：这是不行的，至少要给这个程序输入输出吧。想一想，这个程序的输出应该保存在哪里?

小可：我记得 Hadoop 的两个核心组件是 MapReduce 和 HDFS，HDFS 是用来保存文件的，对于这种大规模的输入量，应该是以文件形式存储的吧，所以应该放在 HDFS 里面。

Mr. 王：没错，我们来看看具体是怎么做的。

首先我们要把程序变成一个 jar 包。

将程序文件放入 Hadoop 文件夹，并且进入 Hadoop 文件夹。

```
~$ cd hadoop-1.0.1
~/hadoop-1.0.1$ mkdir WordCount
```

接下来编译 WordCount.java。

```
~/hadoop-1.0.1 $ javac -classpath ./hadoop-core-1.0.1.jar:lib/commons-cli-1.2.jar:lib/* -d WordCount WordCount.java
```

在这里不同的版本可能会略有不同，形如 hadoop-core-1.0.1.jar 这样的包名可以去 Hadoop 文件夹下查看实际的名字。

打包刚才生成的类文件。

```
~/hadoop-1.0.1$ jar -cvf WordCount.jar -C WordCount .
```

注意不要丢掉最后的点号。

小可：这一步就是把 WordCount 和一些依赖的 Hadoop 部分编译到一起，封装成一个 jar 包。

Mr. 王：接下来我们试试在 Hadoop 框架下运行一下已经编译好的 WordCount。

别忘了格式化 HDFS，使用我们前面说过的命令：

```
bin/hadoop namenode -format
lk@LKPC:~/hadoop-1.0.1$ bin/hadoop namenode -format
15/01/20 19:39:42 INFO namenode.NameNode: STARTUP_MSG:
/************************************************************
STARTUP_MSG: Starting NameNode
STARTUP_MSG:   host = LKPC/127.0.1.1
STARTUP_MSG:   args = [-format]
STARTUP_MSG:   version = 1.0.1
STARTUP_MSG:   build = https://svn.apache.org/repos/asf/hadoop/common/branches/branch-1.0 -r 1243785; compiled by 'hortonfo' on Tue Feb 14 08:15:38 UTC 2012
************************************************************/
Re-format filesystem in /tmp/hadoop-lk/dfs/name ? (Y or N) y
Format aborted in /tmp/hadoop-lk/dfs/name
15/01/20 19:39:44 INFO namenode.NameNode: SHUTDOWN_MSG:
```

```
/************************************************************
SHUTDOWN_MSG: Shutting down NameNode at LKPC/127.0.1.1
************************************************************/
```

屏幕上会出现一些 NameNode 的状态信息。如果出现了是否重新进行格式化的提示，则说明你的 HDFS 已经进行过格式化了，可以重新格式化或者放弃。

格式化完成之后，我们就可以启动 Hadoop 了。使用脚本文件 start-all.sh：

```
~/hadoop-1.0.1$ bin/start-all.sh
```

屏幕上会出现大量的提示信息：

```
starting namenode, logging to /home/lk/hadoop-1.0.1/libexec/../logs/hadoop-lk-namenode-LKPC.out
localhost: starting datanode, logging to /home/lk/hadoop-1.0.1/libexec/../logs/hadoop-lk-datanode-LKPC.out
localhost: starting secondarynamenode, logging to /home/lk/hadoop-1.0.1/libexec/../logs/hadoop-lk-secondarynamenode-LKPC.out
starting jobtracker, logging to /home/lk/hadoop-1.0.1/libexec/../logs/hadoop-lk-jobtracker-LKPC.out
localhost: starting tasktracker, logging to /home/lk/hadoop-1.0.1/libexec/../logs/hadoop-lk-tasktracker-LKPC.out
```

可以看出，Hadoop 依次启动了 NameNode、DataNode、JobTracker 和 TaskTracker，在一些 Hadoop 版本中还有可能包含形如 secondarynamenode 这样的部分。

接下来把输入文件从磁盘放入 HDFS 中。首先我们来看看 HDFS 的常用命令。

可以使用 $ bin/hadoop dfs 命令来查看 HDFS 的命令列表。

```
[-ls <path>]
[-lsr <path>]
[-du <path>]
[-dus <path>]
[-count[-q] <path>]
[-mv <src> <dst>]
[-cp <src> <dst>]
[-rm [-skipTrash] <path>]
[-rmr [-skipTrash] <path>]
[-expunge]
[-put <localsrc> ... <dst>]
[-copyFromLocal <localsrc> ... <dst>]
[-moveFromLocal <localsrc> ... <dst>]
[-get [-ignoreCrc] [-crc] <src> <localdst>]
[-getmerge <src> <localdst> [addnl]]
[-cat <src>]
[-text <src>]
[-copyToLocal [-ignoreCrc] [-crc] <src> <localdst>]
[-moveToLocal [-crc] <src> <localdst>]
[-mkdir <path>]
[-setrep [-R] [-w] <rep> <path/file>]
```

```
[-touchz <path>]
[-test -[ezd] <path>]
[-stat [format] <path>]
[-tail [-f] <file>]
[-chmod [-R] <MODE[,MODE]... | OCTALMODE> PATH...]
[-chown [-R] [OWNER][:[GROUP]] PATH...]
[-chgrp [-R] GROUP PATH...]
[-help [cmd]
```

大部分内容和 Linux Shell 的命令是非常相似的。

我们可以用如下的格式来使用命令：

```
[Hadoop 目录 ]$ bin/hadoop dfs -[ 命令 ] 参数
```

注意不要遗漏命令前面的小横线。

比如，我们要查看 HDFS 中的文件列表，就可以使用如下命令：

```
~/hadoop-1.0.1$ bin/hadoop dfs -ls
```

一般来说，我们要处理的数据量都很大，而且很多时候这些数据往往不止存在一个数据文件中。这里我们用三个文件来举例，多个文件和三个文件的方法是相同的。假设三个文件的名字分别是 file01、file02 和 file03。

首先为它们创建一个文件夹，这个文件夹存储在 HDFS 中。

```
~/hadoop-1.0.1$ bin/hadoop dfs -mkdir input
```

然后将输入文件放进文件夹中。

```
~/hadoop-1.0.1$ bin/hadoop dfs -put ./file01 input
~/hadoop-1.0.1$ bin/hadoop dfs -put ./file02 input
~/hadoop-1.0.1$ bin/hadoop dfs -put ./file03 input
...
```

当文件非常多的时候，我们也可以选择使用通配符“*”。比如：

```
~/hadoop-1.0.1$ bin/hadoop dfs -put ./file* input
```

它用来表示操作对象是所有以 file 开头的文件。使用通配符固然是非常方便的，但是也容易由于使用不当产生一些错误，所以设计文件名时一定要谨慎。

这些都完成之后，我们可以执行前面的 jar 包了。

```
~/hadoop-1.0.1$ bin/hadoop jar WordCount.jar WordCount input output
```

现在我们简单分析一下用 Hadoop 运行 jar 包的命令格式。

```
hadoop jar 包名 .jar 类名 输入路径 输出路径
```

这里的输入路径是我们之前在 HDFS 中创建的 input，output 是我们给输出的日志文件和结果建立的文件夹，它会和 input 一起存放在 HDFS 的根目录下。

小可盯着屏幕上变化的文字，说：从屏幕上可以看到很多运行的信息，还有 Map 和 Reduce 的运行情况百分比。

```
15/01/20 19:42:49 WARN mapred.JobClient: Use GenericOptionsParser for parsing
the arguments. Applications should implement Tool for the same.
```

```
15/01/20 19:42:49 INFO mapred.FileInputFormat: Total input paths to process : 3
15/01/20 19:42:50 INFO mapred.JobClient: Running job: job_201501201941_0001
15/01/20 19:42:51 INFO mapred.JobClient:  map 0% reduce 0%
15/01/20 19:43:05 INFO mapred.JobClient:  map 50% reduce 0%
15/01/20 19:43:11 INFO mapred.JobClient:  map 100% reduce 0%
15/01/20 19:43:14 INFO mapred.JobClient:  map 100% reduce 16%
15/01/20 19:43:23 INFO mapred.JobClient:  map 100% reduce 100%
15/01/20 19:43:28 INFO mapred.JobClient: Job complete: job_201501201941_0001
15/01/20 19:43:28 INFO mapred.JobClient: Counters: 30
15/01/20 19:43:28 INFO mapred.JobClient:   Map-Reduce Framework
15/01/20 19:43:28 INFO mapred.JobClient:     Spilled Records=202
15/01/20 19:43:28 INFO mapred.JobClient:   Map output materialized bytes=1236
15/01/20 19:43:28 INFO mapred.JobClient:     Reduce input records=101
15/01/20 19:43:28 INFO mapred.JobClient: Virtual memory (bytes) snapshot=2757025792
15/01/20 19:43:28 INFO mapred.JobClient:     Map input records=57
15/01/20 19:43:28 INFO mapred.JobClient:     SPLIT_RAW_BYTES=380
15/01/20 19:43:28 INFO mapred.JobClient:     Map output bytes=1010
15/01/20 19:43:28 INFO mapred.JobClient:     Reduce shuffle bytes=1074
15/01/20 19:43:28 INFO mapred.JobClient: Physical memory (bytes) snapshot=748482560
15/01/20 19:43:28 INFO mapred.JobClient:     Map input bytes=606
15/01/20 19:43:28 INFO mapred.JobClient:     Reduce input groups=2
15/01/20 19:43:28 INFO mapred.JobClient:     Combine output records=0
15/01/20 19:43:28 INFO mapred.JobClient:     Reduce output records=2
15/01/20 19:43:28 INFO mapred.JobClient:     Map output records=101
15/01/20 19:43:28 INFO mapred.JobClient:     Combine input records=0
15/01/20 19:43:28 INFO mapred.JobClient:     CPU time spent (ms)=4160
15/01/20 19:43:28 INFO mapred.JobClient: Total committed heap usage (bytes)=767557632
15/01/20 19:43:28 INFO mapred.JobClient:   File Input Format Counters
15/01/20 19:43:28 INFO mapred.JobClient:     Bytes Read=712
15/01/20 19:43:28 INFO mapred.JobClient:   FileSystemCounters
15/01/20 19:43:28 INFO mapred.JobClient:     HDFS_BYTES_READ=1092
15/01/20 19:43:28 INFO mapred.JobClient:     FILE_BYTES_WRITTEN=108978
15/01/20 19:43:28 INFO mapred.JobClient:     FILE_BYTES_READ=1218
15/01/20 19:43:28 INFO mapred.JobClient:     HDFS_BYTES_WRITTEN=18
15/01/20 19:43:28 INFO mapred.JobClient:   File Output Format Counters
15/01/20 19:43:28 INFO mapred.JobClient:     Bytes Written=18
15/01/20 19:43:28 INFO mapred.JobClient:   Job Counters
15/01/20 19:43:28 INFO mapred.JobClient:     Launched map tasks=4
15/01/20 19:43:28 INFO mapred.JobClient:     Launched reduce tasks=1
15/01/20 19:43:28 INFO mapred.JobClient:     SLOTS_MILLIS_REDUCES=16501
15/01/20 19:43:28 INFO mapred.JobClient:       Total time spent by all reduces
waiting after reserving slots (ms)=0
15/01/20 19:43:28 INFO mapred.JobClient:     SLOTS_MILLIS_MAPS=24519
15/01/20 19:43:28 INFO mapred.JobClient:       Total time spent by all maps
```

```
waiting after reserving slots (ms)=0
    15/01/20 19:43:28 INFO mapred.JobClient:     Data-local map tasks=4
```

Mr. 王：是的，当操作的数据比较大时，往往 Map 和 Reduce 运行的时间会非常长，我们可以根据屏幕上输出的这些日志来观察具体的运行情况。关于各种系统资源的使用情况和程序的执行情况，都可以从输出日志中找到相关的统计数据。

小可：可是，现在运行结束之后，结果在哪里呢?

Mr. 王：当发现光标已经重新出现在终端中时，说明 Hadoop 的运行已经结束，我们可以去看看程序的执行结果了。从刚才执行的命令来看，我们将程序的执行结果放在 HDFS 中一个叫 output 的文件夹中了。我们可以先看看这个文件夹是不是存在。

```
    ~/hadoop-1.0.1$ bin/hadoop dfs -ls

    drwxr-xr-x   - lk supergroup          0 2015-01-20 19:41 /user/lk/input
    drwxr-xr-x   - lk supergroup          0 2015-01-20 19:43 /user/lk/output
    ...
```

如果找到了 output，则说明程序已经成功地将结果找了出来。

我们可以将这个文件夹从 HDFS 重新放回硬盘中。可以使用 HDFS 的 -get 命令，这个命令可以将 HDFS 中的文件或文件夹取出来，放到硬盘中。后面的两个参数分别为要操作的对象和要放置的指定路径。

```
    ~/hadoop-1.0.1$ bin/hadoop dfs -get output ./
```

我们再使用 ls 命令，看看是否有这个文件夹就可以了。

```
    bin conf FirstJar hadoop-minicluster-1.0.1.jar  ivy.xml logs.txt sbin
WordCount.java build.xml contrib hadoop-ant-1.0.1.jar hadoop-test-1.0.1.jar
lib NOTICE.txt  share  c++  file01   hadoop-client-1.0.1.jar     hadoop-tools-
1.0.1.jar       libexec      output       src
    CHANGES.txt    file02    hadoop-core-1.0.1.jar        Hadoop_WordCount
LICENSE.txt  Projects    webapps  Combine.java  file03   hadoop-examples-1.0.1.jar
ivy  ogs     README.txt  wordcount.jar
```

我们发现了 output 这个文件夹，进入这个文件夹。

```
    cd output
```

里面会有几个日志文件，其中的 part-00000 就是保存结果的文件，打开它。

```
    vim part-00000
```

显示了每个单词出现的数量。比如：

```
    Hello 23
    World 55
```

Mr. 王：这仍然是一个比较经典版本的 WordCount，在 Hadoop 不断更新和迭代之后，也引入了一些新机制和新 API。下面是一个基于新版本 API 的 WordCount 程序。

```
    1 import java.io.IOException;
    2 import java.util.*;
    3
```

```
 4 import org.apache.hadoop.fs.Path;
 5 import org.apache.hadoop.conf.*;
 6 import org.apache.hadoop.io.*;
 7 import org.apache.hadoop.mapreduce.*;
 8 import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
 9 import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
11 import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
11 import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;
12
13 public class WordCount {
14
 15   public static class Map extends Mapper<LongWritable, Text, Text,
IntWritable> {
16     private final static IntWritable one = new IntWritable(1);
17     private Text word = new Text();
18
 19       public void map(LongWritable key, Text value, Context context)
throws IOException, InterruptedException {
20         String line = value.toString();
21         StringTokenizer tokenizer = new StringTokenizer(line);
22         while (tokenizer.hasMoreTokens()) {
23             word.set(tokenizer.nextToken());
24             context.write(word, one);
25         }
26     }
27  }
28
 29   public static class Reduce extends Reducer<Text, IntWritable, Text,
IntWritable> {
30
 31      public void reduce(Text key, Iterable<IntWritable> values, Context
context)
32       throws IOException, InterruptedException {
33         int sum = 0;
34         for (IntWritable val : values) {
35             sum += val.get();
36         }
37         context.write(key, new IntWritable(sum));
38     }
39  }
40
41  public static void main(String[] args) throws Exception {
42     Configuration conf = new Configuration();
43
44         Job job = new Job(conf, "wordcount");
45
```

```
46      job.setOutputKeyClass(Text.class);
47      job.setOutputValueClass(IntWritable.class);
48
49      job.setMapperClass(Map.class);
50      job.setReducerClass(Reduce.class);
51
52      job.setInputFormatClass(TextInputFormat.class);
53      job.setOutputFormatClass(TextOutputFormat.class);
54
55      FileInputFormat.addInputPath(job, new Path(args[0]));
56      FileOutputFormat.setOutputPath(job, new Path(args[1]));
57
58      job.waitForCompletion(true);
59  }
60
61 }
```

程序的整体设计思路没有太大的变化，在程序的逻辑上也没有很明显的变化，只是在 API 上发生了一点变化。

在新版本的 API 中，我们注意到在 Map 类和 Reduce 类的实现上，不再需要继承 MapReduceBase 类并实现 Mapper 和 Reducer 接口，而是直接继承 Mapper 类和 Reducer 类即可。这说明在新版本的 API 中，Mapper 和 Reducer 已经不再是接口规范，而是一个可供继承的类了。

另外，context 对象也具有新的特点，在很大程度上它代替了 Reporter 的工作。在主方法中，我们也注意到使用 Job 类来控制 Job 的运行，在以往的 API 中，使用 JobClient 类。

小可：关于 WordCount 的内容我基本上懂了。

Mr. 王：别看 WordCount 这个例子比较简单，但真可谓是麻雀虽小，五脏俱全。我们可以用相似的框架结构，以并行计算的方法去完成很多大数据处理的任务。接下来我们来看几个实际的例子。

12.1.4　Hadoop 实践案例——记录去重

Mr. 王：现在我们看一个和 WordCount 很相似，在实际中应用也很多的例子——记录去重。

小可：嗯，从字面上理解就是将重复的数据记录去除吧？

Mr. 王：是的，就是如此。这个工作在实际的应用中是非常常见的，在进行数据管理时，不论是录入记录错误，还是新旧数据的原因，都是非常容易出现重复的记录的。很多时候，重复的记录会对我们进行个数统计等操作产生影响，造成统计结果错误。另外，出现重复记录的数据集合可能会非常大，单靠人工挑重，或者是靠简单的单机去查找会比较慢，所以我们要尝试借助并行机制来解决这个问题。

下面给出一些输入输出的例子。

比如现在有一些通讯录文件：

```
AddressBook1：
Alice 123456789
Bob 234567891
Charlie 345678912
Dean 456789123
AddressBook2：
Edison 123456789
Bob 234567891
Charlie 345678912
Franklin 567891234
```

我们希望合并这些通讯录文件，去掉其中重复的数据，将它变成如下这样不含重复记录的数据：

```
DistinctAddressBook：
Alice 123456789
Bob 234567891
Charlie 345678912
Dean 456789123
Edison 123456789
Franklin 567891234
```

你来想一想这个问题可以怎么解决?

小可：嗯……也不能用每一个数据到整个数据集合里面查找啊，这样效率岂不是太低了。

在很多数据库中，都有很多手段来防止重复数据的出现，比如主键机制。它作为唯一识别数据的标识，是不允许出现重复的。

Mr. 王启发道：想一想之前我们学过哪个例子?

小可顿时恍然大悟：对了，这个工作其实跟 WordCount 很像啊！我们只要对所有的记录进行计数，然后再去掉这个计数就可以了！

Mr. 王：很好，其实仔细想想，记录去重这个工作和 WordCount 是非常相似的。不过记录去重我们可以做的更加简单。第一，我们关注的是记录，而不是单词，所以无须对记录进行单词切分，只需要把整条记录当作一个数据项就可以了。其实这样做也是有必要的，因为很多时候，在电话簿里面具有相同名字的记录并不一定有着相同的电话号码。这就意味着，只有两条完全相同的记录才是重复记录，所以我们使用整条记录作为关键词去重，反而省去了切分单词的操作。第二，在记录去重的工作中，我们并不关心重复记录出现了几次，直接合并它们就可以了，所以完全可以不去设置记录出现数量的计数器。至于在 WordCount 中那个 (word,1) 中的 1，我们随意填写一个数据就可以，一般用空字符串就行，这样比较节省存储空间。

小可：嗯，似乎比 WordCount 更简单了一些。

Mr. 王：好了，我们来看看实现它的源代码。

```
import java.io.IOException;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.util.*;

public class DataDistinct
{
    public static class Map extends Mapper <Object,Text,Text,Text>
    {
        private static Text record_text = new Text();
        public void map(Object key,Text value,Context context) throws
IOException,InterruptedException
        {
            record_text = value;
            context.write (record_text,new Text(""));
        }
    }
    public static class Reduce extends Reducer <Text,Text,Text,Text>
    {
        public void reduce(Text key,Iterable<Text> values,Context context)
throws IOException,InterruptedException
        {
            context.write(key,new Text(""));
        }
    }
    public static void main(String[] args) throws Exception
    {
        Configuration conf = new Configuration();
        String[] Arguments = new GenericOptionsParser(conf,args).
getRemainingArgs();
        if (Arguments.length != 2)
        {
            System.err.println("Error in Arguments,please set input and
output");
            System.exit(2);
        }
```

```
        Job job = new Job (conf,"DataDistinct");
        job.setJarByClass (DataDistinct.class);
        job.setMapperClass (Map.class);
        job.setReducerClass (Reduce.class);
        job.setOutputKeyClass (Text.class);
        job.setOutputValueClass (Text.class);
        FileInputFormat.addInputPath (job,new Path (Arguments[0]));
        FileOutputFormat.setOutputPath (job,new Path (Arguments[1]));
        System.exit (job.waitForCompletion(true) ? 0:1);
    }
}
```

Mr. 王：前面和后面的部分这里就不多说了，可以看作是实现类似的 Hadoop 操作的固定格式。当然，其实其中的内容也是非常容易识别的，Java 以完整的单词进行类和对象命名的特点使得类和操作的识别变得非常容易，可以很容易地读懂。我们重点来看看对 Map 和 Reduce 两个基本操作的设计。

首先来看 Map 的设计。

在新版本 Hadoop 的标准 API 中，Map 是 Mapper 基类的派生类。我们先定义一个派生于 Mapper 的 Map 类：

```
public static class Map extends Mapper <Object,Text,Text,Text>
```

然后定义一个 map 函数，设计接收的 key-value 对的形式：

```
public void map(Object key,Text value,Context context) throws IOException,Int
erruptedException
```

从接收到的来自文本的数据记录的内容提取出来作为一个新的变量：

```
record_text = value;
```

在新版本的 API 中，我们使用 context 来表示要发出（emit）的数据记录。在这里我们将接收到的数据作为 key，而 value 就像前面说过的那样，填写一个空值就可以了：

```
context.write (record_text,new Text(""));
```

接下来是 reduce 类，它同样派生于 Reducer 基类：

```
public static class Reduce extends Reducer <Text,Text,Text,Text>
```

在 reduce 这个函数中，我们定义 Reduce 的基本操作，在这里要接收 map 发出的键值对。虽然 value 并没有意义，但我们依然要像有 value 那样生成一个 values 列表来接收大量的空串：

```
public void reduce(Text key,Iterable<Text> values,Context context) throws IOE
xception,InterruptedException
```

不过在这个 Reduce 中，我们并不执行任何操作，只是将接收到的 key 写到结果中：

```
context.write(key,new Text(""));
```

12.1.5 Hadoop 实践案例——等值连接

Mr. 王：我们再来看看另一个非常常见的例子。很多时候，我们关心的数据来自多个表。

比如在某学校的教务系统中，有学号和学生姓名的关系表。

```
001 Alice
002 Bob
003 Charlie
004 Dean
```

表中的两列分别是学号和姓名。

小可：嗯，这在教务系统中是很常见的表格。

Mr. 王：而在期末时需要保存学生的成绩单，这个成绩单是学号和成绩的对应表。

```
001 100
002 99
003 77
004 88
```

Mr. 王：表中的两列分别是学号和成绩。但这样的表格在数据库系统中虽然非常常见，但是用户读起来却非常不直观。我们希望看到的是学号、姓名和成绩的表。

```
001 Alice  100
002 Bob  99
003 Charlie  77
004 Dean  88
```

Mr. 王：要实现这个功能就需要用到等值连接，等值连接进行的操作就是将两个表中在相同属性上具有相同值的记录连接起来。这种操作在很多数据库系统中都有实现，是一种非常有价值的操作。

小可：这个问题应该如何解决呢?

Mr. 王：这里还是要联想到我们做过的最基本的例子：WordCount。仔细想一想，这个操作和 WordCount 是不是也有相似之处呢?

小可回想了一下前面的程序，点点头：的确是啊。这里的 key 就是两个表所具有的相同属性，其他属性就是 value，Map 函数可以分条接收表中的记录。在 Reduce 时，Hadoop 会自动将在 key 上具有相同的值，也就是两个表的相同属性上具有相同值的记录聚集在一起，然后将它们的 value 连接起来就可以了!

Mr. 王笑着说：很好，看来你已经具有一定的以 MapReduce 的思维思考问题的能力了。这一次，我希望你自己来试试写一下这个程序。

小可：好的，我自己来试试。为了方便起见，在这里我暂时只考虑两个表仅有一个相同的属性，而且两个表中的其他属性只有一列的情况。将这种情况扩展成多列的情况其实非常容易，只要将那些属性组合起来，形成长串或者数组就可以了。

Mr. 王：好的。

小可打开计算机，在上面敲了一会儿。

小可舒了一口气：写好了!

```
import java.io.IOException;
import java.util.*;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.util.*;

public class NaturalJoin
{
    public static class Map extends Mapper <Object,Text,Text,Text>
    {
            private static Text record_text = new Text();
            public void map(Object key,Text value,Context context) throws IOEx
ception,InterruptedException
            {
                StringTokenizer tk = new StringTokenizer (value.toString());
                // 引入 StringTokenizer 对 Map 接收到的行进行分割，形成各个数据项
                String keystr = new String(tk.nextToken());
                String valuestr = new String(tk.nextToken());
                // 这里处理的表格仅有两列，对于多列的情况可以进行一些简单的组合处理
                context.write (new Text (keystr),new Text(valuestr));
            }
    }
    public static class Reduce extends Reducer <Text,Text,Text,Text>
    {
            public void reduce(Text key,Iterable<Text> values,Context context)
throws IOException,InterruptedException
            {
                //reduce 函数将在 key 上具有相同值的那些记录聚集到一起
                Iterator it = values.iterator();
                String value1=it.next().toString();
                String value2=it.next().toString();

                String value = new String();
                String blank = new String(" ");
                if (value1.charAt(0) <='0' && value.charAt(1) <= '9')
                // 这里使用了一个小技巧，通过数据类型（是字母还是数字）来判断它来自表 1
                // 还是表 2
                {
```

```
                    value = value2 + blank + value1;
                    // 将串组合起来，形成输出格式
                }
                else
                {
                    value =value1 + blank + value2;
                }
                context.write(key,new Text(value));

            }
        }

        public static void main(String[] args) throws Exception
        {
            Configuration conf = new Configuration();
            String[] Arguments = new GenericOptionsParser(conf,args).
getRemainingArgs();
            if (Arguments.length != 2)
            {
                System.err.println("Error in Arguments,please set input and
output");
                System.exit(2);
            }
            Job job = new Job (conf,"SimpleNatualJoin");
            job.setJarByClass (NaturalJoin.class);
            job.setMapperClass (Map.class);
            job.se tReducerClass (Reduce.class);
            job.setOutputKeyClass (Text.class);
            job.setOutputValueClass (Text.class);
            FileInputFormat.addInputPath (job,new Path (Arguments[0]));
            FileOutputFormat.setOutputPath (job,new Path (Arguments[1]));
            System.exit (job.waitForCompletion(true) ? 0:1);
        }
    }
```

Mr. 王看了看小可书写的代码，说：好，用刚才的例子运行一下试试。

经过了前面的学习，小可熟练地操作启动 Hadoop 服务，并完成了代码的编译。

小可盯着屏幕上的结果：成功了！完全符合我们想要的结果——将两个表合并成了一个表，并且将在相同属性上具有相同值的那些记录合并成了一条。

小可看了看老师，他的脸上好像没有浮现出完全满意的笑容。

小可：程序有哪里不对吗？

Mr. 王：对于这个例子，你的程序完全可以实现了。不过对于学生成绩这种数据，有一个

很重要的特点，那就是在相同属性学号上面，每个学号都是唯一的。不过对于一些其他的例子，数据可就不这么友好了。比如：

表 1　信件邮编表

信件代号	信件邮编
1	150000
2	100000
3	150000
4	200000
5	100000

表 2　邮编表

100000	北京市
200000	上海市
150000	哈尔滨市

我们希望将信件和它的目的地对应起来，结果就是：

1	哈尔滨市
2	北京市
3	哈尔滨市
4	上海市
5	北京市

或者

1	哈尔滨市	150000
2	北京市	100000
3	哈尔滨市	150000
4	上海市	200000
5	北京市	100000

小可：嗯，这样的例子的确也是非常常见的，第二个表相当于一个查找表。而我们需要进行连接的那一列的每一个数据却不是唯一的，可能是多对一或者多对多的情况。这样我的那个程序的确会出现问题。我只考虑了一对一连接的情况，也就是在另一个表中，只存在唯一的在相同属性上与这个表中相同的记录。

Mr. 王：的确是这样，虽然形如学号的这种情况，表中连接属性的每一个值唯一也是非常常见的，你的程序对于这一类情况是可以的。但是我们在设计程序时，还是要考虑到各种不同的情况。

小可：Map 部分应该可以不用修改，只是 Reduce 需要做一点小小的改动就可以了。

Mr. 王：是的，这里我给你提供一个思路，回去自己尝试一下哦。首先，在 Reduce 函数那里，

我们已经完成了在相同属性上具有相同值的筛选。

对于刚才的例子，我们在 value 上接收到的会是：

```
key = 150000
value:
哈尔滨
1
3
```

这样的数据。

所以我们可以在 Reduce 函数中建立两个数组，分别用于存放来自两个表的数据记录。

```
String sheet1[] = new String[SIZE];
String sheet2[] = new String[SIZE];
```

在判断数据究竟来自哪个表的问题上，我们可以使用一些小小的技巧，比如通过数据类型进行判断。通过对接收到的 value 进行一次扫描就可以完成。

```
Iterator it = values.iterator();
while (it.hasNext())
{
    // 判断这些记录实际来自表 1 还是表 2
}
```

其实接下来的操作就非常容易了，只要写一个双层循环，让来自两个表的 value 进行组合就可以了。

```
for (int i=0 ;i < Sheet1 的数据个数 ; i++)
    for (int j=0 ;j < Sheet2 的数据个数 ; j++)
            context.write (new Text(Sheet1[i]),new Text(Sheet2[j]))
    // 这里也可以替换成其他你需要的数据格式
```

小可：哈哈，老师已经几乎把这个程序写出来了。

Mr. 王：嗯，将其合并成一个完整程序的工作就交给你了！（这里留给读者去实现一下，将前面小可的程序，改成可以处理重复值的等值连接程序）

小可：好的！

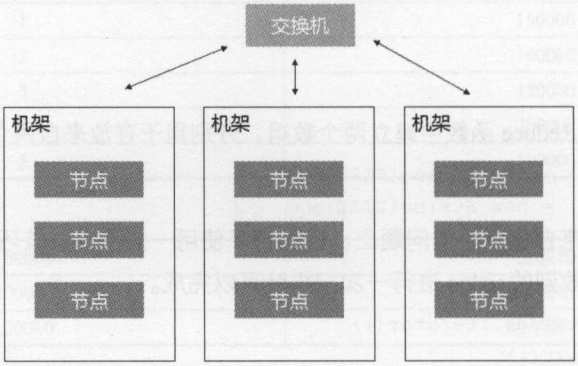
Mr. 王总结道：不难看出，我们实现的这几种操作都万变不离其宗，每一种操作都可以从 WordCount 这个最简单的程序演化而来。其实可以看出，MapReduce 实现的一个非常核心的操作就是将我们关注的数据在 Map 阶段使其出现在 key 上，然后在 Reduce 阶段将那些在 key 上具有相同值的记录聚集在一起，最后对 value 进行合并、筛选或者其他处理。很多 MapReduce 的操作，我们都是可以利用这样的原理完成的。

12.1.6　多机配置

Mr. 王：在关于 Hadoop 内容介绍的最后，我们来谈谈如何把 Hadoop 配置在多台计算机上。

小可：嗯，到目前为止，所有的程序还都仅仅运行在一台计算机上。

Mr. 王：在实际的 Hadoop 应用中，我们使用的往往是多台服务器组成的服务器阵列。有时还要使用多个机架，这些机架之间通过一个交换机相连，每个机架的内部还有多台服务器核心，这些服务器之间也通过交换机相连接。当我们要处理的数据量达到一定规模时，每个机架中会有几十台计算机参与到并行计算之中。



这里我们不介绍几十台计算机的情况，那样叙述起来太过麻烦，我们用只有 3 个节点的情况来描述，其实在基本配置上机器比较多的情况和 3 台机器的情况是非常类似的。不过在真正的 Hadoop 应用配置上，还有很多需要注意的问题，比如网络拓扑、守护进程、输入输出缓冲等，这里暂时不作介绍。

首先要把准备工作全都做好，这些准备工作和配置伪分布式环境是一样的，安装 Linux、安装 JDK、安装 SSH、下载和解压缩 Hadoop。

在伪分布式环境中，仅有的一台机器同时充当 Master 和 Slave、NameNode 和 DataNode、JobTracker 和 TaskTracker。但在完全分布式环境下，这些角色必须确定。虽然 Hadoop 可以自动指定由哪些计算机来执行 Map 和 Reduce，但是上述这些角色还是需要我们手动确定的。我们先对这个分配做一个小小的计划。

IP 地址	主机名	角色
xxx.xxx.xxx.1	M	Master，NameNode，JobTracker
xxx.xxx.xxx.2	S1	Slave，DataNode，TaskTracker
xxx.xxx.xxx.3	S2	Slave，DataNode，TaskTracker

小可：嗯，我们把主机 M 设定为管理系统、检测整个工作、管理 HDFS 的机器，然后将 S1 和 S2 分别设定为具体执行工作的“工人”。

Mr. 王：接下来我们要让每台计算机都知道自己的主机名和 IP 地址。在 Linux 系统下，我们找到 /etc/hosts 这个文件，在里面加上：

```
127.0.0.1 localhost
xxx.xxx.xxx.1 M
xxx.xxx.xxx.2 S1
```

```
xxx.xxx.xxx.3 S2
```

localhost 表示本地，本地的 IP 地址为 127.0.0.1。如果文件中有这个项目的话，不必更改它。

再找到 /etc/hostname 文件：

- 在 M 机器中，就在文件中输入 M。
- 在 S1（S2）机器中，就在文件中输入 S1（S2）。

接下来我们让计算机之间可以通过 SSH 互相免密码登录。关于如何生成 SSH 公密钥对我们前面已经解释过了，方法是一样的，只是这次我们要将 authorized_keys 这个文件夹复制到 S1、S2 中的 .ssh/ 文件夹里面去。

可以使用命令：

```
scp authorized_keys [主机名 如 S1]:~/.ssh/
```

然后试一下在主机上能不能登录到 S1 和 S2。

```
ssh [主机名 如 S1]
```

接下来修改 3 台机器的 Hadoop 配置文件。

对于 3 台计算机，首先在 Hadoop-env.sh 中添加 JAVA_HOME 环境变量。在文件中添加：

```
export JAVA_HOME = [JDK 安装路径 ]/[JDK 文件夹名 ]
```

然后修改配置文件 core-site.xml。

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>

<!-- Put site-specific property overrides in this file. -->

<configuration>
    <property>
            <name>fs.default.name</name>
            <value>hdfs://M:9000</value>
    </property>
<property>
            <name>hadoop.tmp.dir</name>
            <value>/tmp</value>
    </property>
</configuration>
```

这一步我们编辑的是 Hadoop 的配置文件，在这里要指定 HDFS 的地址和端口号。

再修改 MapReduce 的配置文件。

```
<configuration>
    <property>
        <name>mapred.job.tracker</name>
        <value>M:9001</value>
    </property>
</configuration>
```

这里我们要对 MapReduce 的 JobTracker 的地址和端口号进行配置。

最后修改配置文件 hdfs-site.xml。

```
<configuration>
    <property>
        <name>dfs.replication</name>
        <value>2</value>
    </property>
</configuration>
```

这里我们修改的是 HDFS 的配置文件，要确定运行模式为完全分布式并且有两台计算机参与到实际的工作中来。

接下来在 conf/masters 中加入主机名 M，在 conf/slaves 中加入主机名 S1、S2。

在文件中写上：

```
S1
S2
```

至此，一个由 3 个节点组成的 Hadoop 平台就已经构建完毕了。我们可以格式化 HDFS 并启动 Hadoop 了，与伪分布式一样，使用命令：

```
$ bin/Hadoop NameNode -format
$ bin/start-all.sh
```

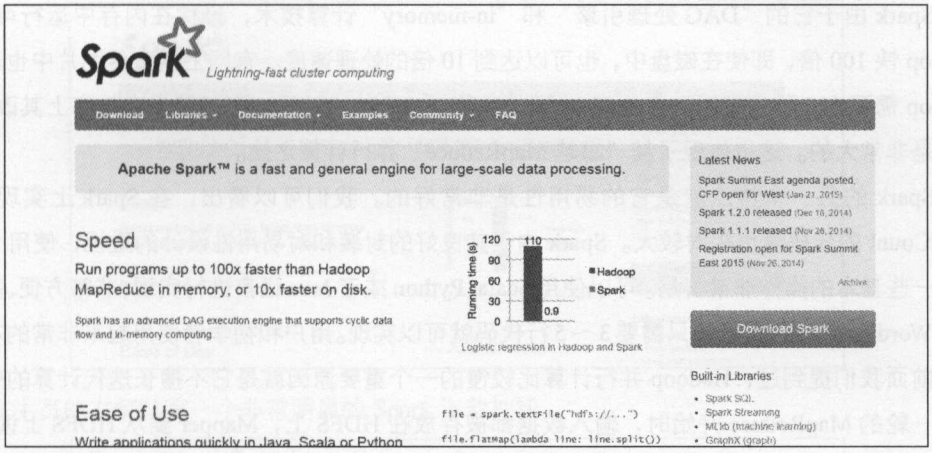
12.2 适于迭代并行计算的平台——Spark

12.2.1 Spark 初探

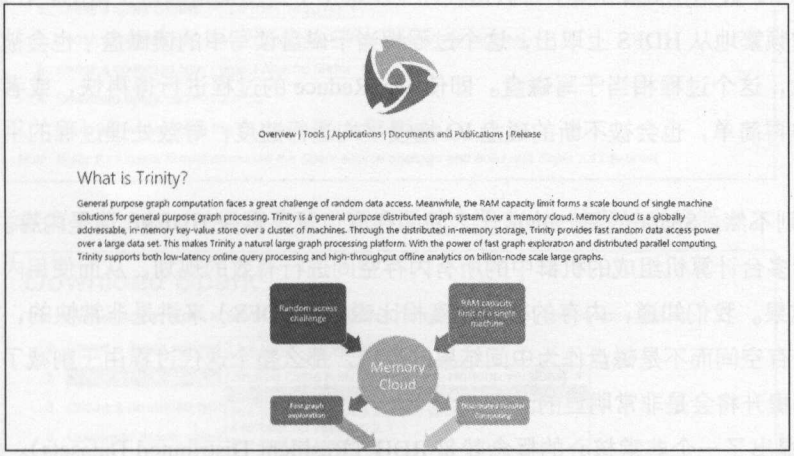
Mr. 王：在初步了解了并行平台 Hadoop 的使用之后，我们再来尝试使用一个超越 MapReduce 的并行平台——Spark。

小可：前面我好像听到过这个名字。

Mr. 王：我们在讨论超越 MapReduce 并行计算时曾经提到过，在以 Hadoop 为代表的 MapReduce 被提出时，在企业级别的应用上 MapReduce 的表现还是可圈可点的，但随着对大数据处理时间要求的不断苛刻和计算复杂程度的提升，人们逐渐地发现了 MapReduce 的各种缺陷，也在寻求着各种改进 MapReduce 的方法，或者提出一些新的模型。在 MapReduce 逐渐被研究人员放弃的时代，大量新平台的出现也让我们眼前一亮，像 Spark 和 Trinity 这样的新一代大数据并行计算平台就是这个时代的产物，它们各有特点，在各自着重注意的一些方面上，它们的表现要比 MapReduce 更加出色。



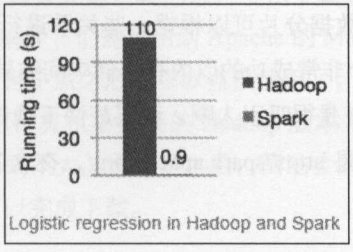
Apache Spark 官方网站



微软研究院 Trinity 官方网站

在这里我们就以非常友好、简单、易用的 Spark 平台为例，来了解一下如何使用新兴的并行大数据平台。Spark 是一个开源的分布式计算平台系统，它存在的目的就是为了提升计算效率。

在 Spark 官方网站上，就展示了 Spark 的很多优点。



Spark 由于它的“DAG 处理引擎”和“in-memory”计算技术，号称在内存中运行可以比 Hadoop 快 100 倍，即使在磁盘中，也可以达到 10 倍的处理速度。官网上的宣传图片中也表示，Hadoop 需要 110 秒的逻辑回归处理，Spark 只需要 0.9 秒的时间，可以看出在效率上其改进力度还是非常大的。这也体现了其“超越 MapReduce”并行计算之处。

Spark 的另一个特点就是它的易用性是非常好的。我们可以看出，在 Spark 上实现一个 WordCount 的代码量也相对较大。Spark 由于其良好的封装和对易用性设计的重视，使用 Spark 实现一些基本的操作非常容易。可以使用 Scala、Python 或者 Java 语言进行实现，非常方便、简洁。就拿 Word Count 来说，可能只需要 3～5 行代码就可以实现。用户和初学者使用起来非常的友好。

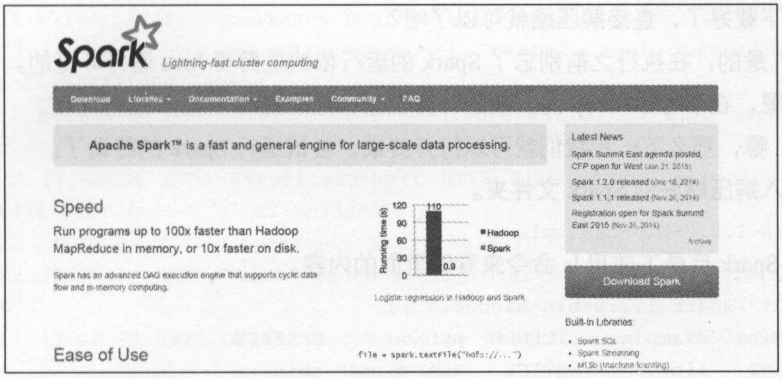
前面我们提到过，Hadoop 并行计算比较慢的一个重要原因就是它不擅长迭代计算的处理。在每一轮的 MapReduce 开始时，输入数据都被存放在 HDFS 上，Mapper 要从 HDFS 上读取数据，处理后送给 Reduce，结果仍然会被保存在 HDFS 上。不过，如果这个过程要进行多个轮次，比如做图算法、数据挖掘算法等，那么迭代几十次甚至上百次都是非常正常的。在这种情况下，数据就会被频繁地从 HDFS 上取出，这个过程相当于磁盘读写中的读磁盘；也会被频繁地存储到 HDFS 上，这个过程相当于写磁盘。即使 MapReduce 的过程进行得再快，或者 MapReduce 执行的操作再简单，也会被不断的磁盘 IO 拖慢平均运行速度，导致处理过程的平均效率大大下降。

Spark 则不然，Spark 的每一轮迭代之间的存取位置不再是 HDFS，而是内存。Spark 非常有效地利用多台计算机组成的机群中的所有内存空间进行有效的规划，从而使用内存来存储所有的中间结果。我们知道，内存的存取速度相比磁盘（HDFS）来讲是非常快的，如果能够有效地利用内存空间而不是磁盘作为中间结果的存储，那么整个迭代过程由于削减了巨大的磁盘开销，效率提升将会是非常明显的。

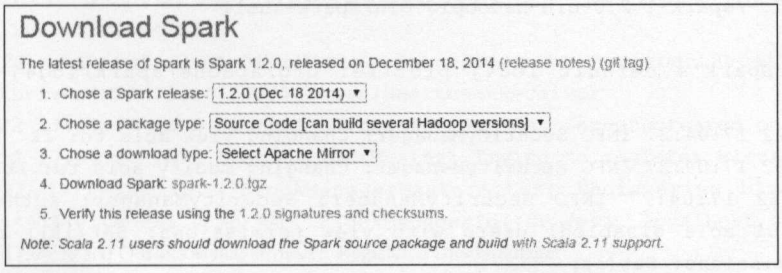
Spark 提出了一个非常核心的概念就是 RDD（Resilient Distributed Datasets）。翻译过来就是弹性分布式数据集合。有了它，Spark 就可以实现 in-memory 的机群级别的并行数据处理。RDD 实现了对数据的分片，对于一个比较大的数据集合，Spark 会将它们分成具有固定大小的分片（就像磁盘中的盘块），这样更加有利于对数据的处理。而且对于每个分片，Spark 都会给出一个函数去处理它，这就相当于一个个小的数据节点，并且每个数据节点都会按照自己应该执行的动作去执行。而且这些数据分片可以根据一些关系进行变换成为新的 RDD。这些新兴的思想都使得 Spark 成为了一个非常成功的以内存存储中间结果的并行平台。

小可兴奋地说：听起来还真是很吸引人啊，我要赶快下载试试。

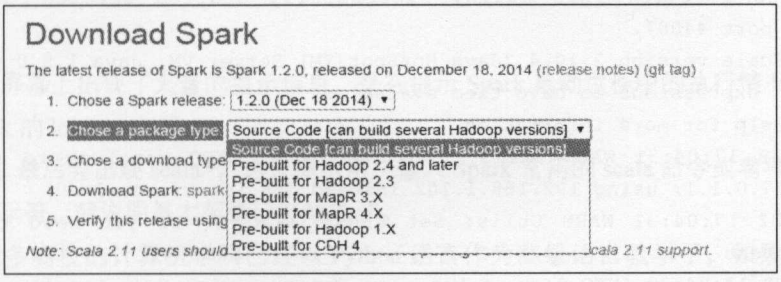
Mr. 王：Spark 的官方网站是 http://spark.apache.org/，在这上面可以找到 Spark 的下载文件和相关文档。



在主页的右侧就有一个非常明显的 Spark 下载按钮。



打开之后就有下载 Spark 的各种选项，我们可以选择 Spark 的各种发行版本。



这里需要注意的一点是，在 Chose a package type: 选项后面，可以看到有很多的下载包选项。为了方便起见，这里选择使用它的预编译版本，也就是

前面带有 Pre-built 的版本。如果你感兴趣的话，可以下载源代码版本，Spark 的源代码非常小，只有几十 MB，不过想要编译它们需要用到 Apache 的 Maven 工具，这里我就不赘述了。预编译好的版本相对要大一些，但下载之后就可以直接使用了，非常的方便。

下载时，我们可以根据自己计算机上安装的 Hadoop 版本来下载安装相应的预编译版本，在我们讨论的范围内，版本的影响不大。选好了之后，网站会向我们推荐下载源，也就是距离我们最近的镜像，跟着提示就可以完成下载。

小可：下载好了，直接解压缩就可以了吧?

Mr. 王：是的，在执行之前别忘了 Spark 的运行依然是需要 Java 运行环境的。

小可：嗯，在学习 Hadoop 时，我的计算机里已经配置好了 Java 运行环境。

Mr. 王：嗯，那么下一步我们就可以打开终端，尝试运行 Spark 的终端了。

首先进入解压缩好的 Spark 文件夹。

```
cd spark-1.2.0-bin-hadoop1/
```

然后在 Spark 目录下使用 ls 命令来看看里面的内容。

```
lk@LKPC:~/spark-1.2.0-bin-hadoop1$ ls
bin   data  examples  LICENSE  python     RELEASE
conf  ec2   lib       NOTICE   README.md  sbin
```

Spark 的执行文件在 bin 中，我们可以使用下面的命令来执行它。

```
lk@LKPC:~/spark-1.2.0-bin-hadoop1$ bin/spark-shell

    Using Spark's default log4j profile: org/apache/spark/log4j-defaults.
properties
    15/01/22 17:04:27 INFO SecurityManager: Changing view acls to: lk
    15/01/22 17:04:27 INFO SecurityManager: Changing modify acls to: lk
    15/01/22 17:04:27 INFO SecurityManager: SecurityManager: authentication
disabled; ui acls disabled; users with view permissions: Set(lk); users with
modify permissions: Set(lk)
    15/01/22 17:04:27 INFO HttpServer: Starting HTTP Server
    15/01/22 17:04:27 INFO Utils: Successfully started service 'HTTP class
server' on port 44087.
    Using Scala version 2.10.4 (Java HotSpot(TM) Server VM, Java 1.8.0_05)
    Type in expressions to have them evaluated.
    Type :help for more information.
    15/01/22 17:04:31 WARN Utils: Your hostname, LKPC resolves to a loopback
address: 127.0.1.1; using 192.168.1.102 instead (on interface wlan0)
    15/01/22 17:04:31 WARN Utils: Set SPARK_LOCAL_IP if you need to bind to
another address
    15/01/22 17:04:32 INFO SecurityManager: Changing view acls to: lk
    15/01/22 17:04:32 INFO SecurityManager: Changing modify acls to: lk
    15/01/22 17:04:32 INFO SecurityManager: SecurityManager: authentication
disabled; ui acls disabled; users with view permissions: Set(lk); users with
modify permissions: Set(lk)
    15/01/22 17:04:32 INFO Slf4jLogger: Slf4jLogger started
    15/01/22 17:04:32 INFO Remoting: Starting remoting
    15/01/22 17:04:33 INFO Remoting: Remoting started; listening on addresses
:[akka.tcp://sparkDriver@LKPC.local:38791]
    15/01/22 17:04:33 INFO Utils: Successfully started service 'sparkDriver' on
port 38791.
    15/01/22 17:04:33 INFO SparkEnv: Registering MapOutputTracker
```

```
15/01/22 17:04:33 INFO SparkEnv: Registering BlockManagerMaster
15/01/22 17:04:33 INFO DiskBlockManager: Created local directory at /tmp/spark-local-20150122170433-aeee
15/01/22 17:04:33 INFO MemoryStore: MemoryStore started with capacity 265.0 MB
15/01/22 17:04:34 INFO HttpFileServer: HTTP File server directory is /tmp/spark-21b3647a-880f-4e24-b797-e37eb95d491f
15/01/22 17:04:34 INFO HttpServer: Starting HTTP Server
15/01/22 17:04:34 INFO Utils: Successfully started service 'HTTP file server' on port 48636.
15/01/22 17:04:39 INFO Utils: Successfully started service 'SparkUI' on port 4040.
15/01/22 17:04:39 INFO SparkUI: Started SparkUI at http://LKPC.local:4040
15/01/22 17:04:39 INFO Executor: Using REPL class URI: http://192.168.1.102:44087
15/01/22 17:04:39 INFO AkkaUtils: Connecting to HeartbeatReceiver: akka.tcp://sparkDriver@LKPC.local:38791/user/HeartbeatReceiver
15/01/22 17:04:39 INFO NettyBlockTransferService: Server created on 54474
15/01/22 17:04:39 INFO BlockManagerMaster: Trying to register BlockManager
15/01/22 17:04:39 INFO BlockManagerMasterActor: Registering block manager localhost:54474 with 265.0 MB RAM, BlockManagerId(<driver>, localhost, 54474)
15/01/22 17:04:39 INFO BlockManagerMaster: Registered BlockManager
15/01/22 17:04:40 INFO SparkILoop: Created spark context..
Spark context available as sc.

scala>
```

小可：屏幕上出现了大量的提示信息，是在提示 Spark 启动过程中的执行情况吧。中间还有一个 Spark 的 logo！

Mr. 王：最后会出现 scala>，这是提示用户输入 Spark 常用的 scala 命令或者程序。如果出现了这个提示符，就说明基本配置已经成功了。

现在很多高校的计算机学科已经以 Python 语言作为高级语言教学了，如果你比较擅长 Python 的话，也可以用 Spark 提供以 Python 为基础语言的终端。使用命令：

```
lk@LKPC:~/spark-1.2.0-bin-hadoop1$ bin/pyspark
Python 2.7.3 (default, Aug  1 2012, 05:16:07)
[GCC 4.6.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
15/01/22 17:07:25 WARN Utils: Your hostname, LKPC resolves to a loopback address: 127.0.1.1; using 192.168.1.102 instead (on interface wlan0)
15/01/22 17:07:25 WARN Utils: Set SPARK_LOCAL_IP if you need to bind to another address
```

```
    15/01/22 17:07:26 INFO SecurityManager: Changing view acls to: lk
    15/01/22 17:07:26 INFO SecurityManager: Changing modify acls to: lk
    15/01/22 17:07:26 INFO SecurityManager: SecurityManager: authentication
disabled; ui acls disabled; users with view permissions: Set(lk); users with
modify permissions: Set(lk)
    15/01/22 17:07:26 INFO Slf4jLogger: Slf4jLogger started
    15/01/22 17:07:26 INFO Remoting: Starting remoting
    15/01/22 17:07:26 INFO Remoting: Remoting started; listening on addresses
:[akka.tcp://sparkDriver@LKPC.local:47691]
    15/01/22 17:07:26 INFO Utils: Successfully started service 'sparkDriver' on
port 47691.
    15/01/22 17:07:26 INFO SparkEnv: Registering MapOutputTracker
    15/01/22 17:07:26 INFO SparkEnv: Registering BlockManagerMaster
    15/01/22 17:07:26 INFO DiskBlockManager: Created local directory at /tmp/
spark-local-20150122170726-9da1
    15/01/22 17:07:26 INFO MemoryStore: MemoryStore started with capacity 265.0
MB
    15/01/22 17:07:26 INFO HttpFileServer: HTTP File server directory is /tmp/
spark-2a7af27a-51a9-47fa-92d7-68e80eb2fa5e
    15/01/22 17:07:26 INFO HttpServer: Starting HTTP Server
    15/01/22 17:07:26 INFO Utils: Successfully started service 'HTTP file server'
on port 43988.
    15/01/22 17:07:31 INFO Utils: Successfully started service 'SparkUI' on port
4040.
    15/01/22 17:07:31 INFO SparkUI: Started SparkUI at http://LKPC.local:4040
    15/01/22 17:07:32 INFO AkkaUtils: Connecting to HeartbeatReceiver: akka.
tcp://sparkDriver@LKPC.local:47691/user/HeartbeatReceiver
    15/01/22 17:07:32 INFO NettyBlockTransferService: Server created on 41433
    15/01/22 17:07:32 INFO BlockManagerMaster: Trying to register BlockManager
    15/01/22 17:07:32 INFO BlockManagerMasterActor: Registering block manager
localhost:41433 with 265.0 MB RAM, BlockManagerId(<driver>, localhost, 41433)
    15/01/22 17:07:32 INFO BlockManagerMaster: Registered BlockManager
    Welcome to
    Using Python version 2.7.3 (default, Aug  1 2012 05:16:07)
    SparkContext available as sc.
    >>>
```

如果最后出现了“>>>”符号，则说明 Python 终端已经顺利启动了。

12.2.2 单词出现行计数

Mr. 王：我们可以试试用 Python 终端来实现一个最简单的功能——单词出现行计数。

首先创建一个文件，在里面写一段话。

小可：我就在 Spark 文件夹里写一个名为 HelloWorld 的文件吧！

```
Hello World!
I am a rookie of Spark.
Now I am trying the Hello World application in Spark!
Have fun!
```

Mr. 王：好，我们现在就让 Spark 来执行一个在文本处理中非常简单却非常常用的功能。首先求出整个文本文件有多少行，然后求出有某个关键词出现的行数，为进行其他处理打下基础。

首先加载 HelloWorld 文件，使用命令：

```
>>> textFile = sc.textFile("HelloWorld")
```

程序会有一些输出，显示程序的运行情况。

```
15/01/22 17:15:33 WARN SizeEstimator: Failed to check whether
UseCompressedOops is set; assuming yes
15/01/22 17:15:33 INFO MemoryStore: ensureFreeSpace(28887) called with
curMem=0, maxMem=277877882
15/01/22 17:15:33 INFO MemoryStore: Block broadcast_0 stored as values in
memory (estimated size 28.2 KB, free 265.0 MB)
15/01/22 17:15:34 INFO MemoryStore: ensureFreeSpace(4959) called with
curMem=28887, maxMem=277877882
15/01/22 17:15:34 INFO MemoryStore: Block broadcast_0_piece0 stored as bytes
in memory (estimated size 4.8 KB, free 265.0 MB)
15/01/22 17:15:34 INFO BlockManagerInfo: Added broadcast_0_piece0 in memory
on localhost:54695 (size: 4.8 KB, free: 265.0 MB)
15/01/22 17:15:34 INFO BlockManagerMaster: Updated info of block broadcast_0_
piece0
15/01/22 17:15:34 INFO SparkContext: Created broadcast 0 from textFile at
NativeMethodAccessorImpl.java:-2
```

然后通过查看文件的第一行，看看是不是正确地加载了这个文件。输入下面的命令：

```
>>> textFile.first()
15/01/22 17:16:17 INFO SparkContext: Starting job: runJob at PythonRDD.
scala:344
15/01/22 17:16:17 INFO DAGScheduler: Got job 1 (runJob at PythonRDD.
scala:344) with 1 output partitions (allowLocal=true)
15/01/22 17:16:17 INFO DAGScheduler: Final stage: Stage 1(runJob at
PythonRDD.scala:344)
15/01/22 17:16:17 INFO DAGScheduler: Parents of final stage: List()
15/01/22 17:16:17 INFO DAGScheduler: Missing parents: List()
15/01/22 17:16:17 INFO DAGScheduler: Submitting Stage 1 (PythonRDD[3] at RDD
at PythonRDD.scala:43), which has no missing parents
15/01/22 17:16:17 INFO MemoryStore: ensureFreeSpace(4576) called with
curMem=43214, maxMem=277877882
15/01/22 17:16:17 INFO MemoryStore: Block broadcast_2 stored as values in
memory (estimated size 4.5 KB, free 265.0 MB)
```

```
15/01/22 17:16:17 INFO MemoryStore: ensureFreeSpace(3392) called with
curMem=47790, maxMem=277877882
15/01/22 17:16:17 INFO MemoryStore: Block broadcast_2_piece0 stored as bytes
in memory (estimated size 3.3 KB, free 265.0 MB)
15/01/22 17:16:17 INFO BlockManagerInfo: Added broadcast_2_piece0 in memory
on localhost:54695 (size: 3.3 KB, free: 265.0 MB)
15/01/22 17:16:17 INFO BlockManagerMaster: Updated info of block broadcast_2_
piece0
15/01/22 17:16:17 INFO SparkContext: Created broadcast 2 from broadcast at
DAGScheduler.scala:838
15/01/22 17:16:17 INFO DAGScheduler: Submitting 1 missing tasks from Stage 1
(PythonRDD[3] at RDD at PythonRDD.scala:43)
15/01/22 17:16:17 INFO TaskSchedulerImpl: Adding task set 1.0 with 1 tasks
15/01/22 17:16:17 INFO TaskSetManager: Starting task 0.0 in stage 1.0 (TID 2,
localhost, PROCESS_LOCAL, 1313 bytes)
15/01/22 17:16:17 INFO Executor: Running task 0.0 in stage 1.0 (TID 2)
15/01/22 17:16:17 INFO HadoopRDD: Input split: file:/home/lk/spark-1.2.0-bin-
hadoop1/HelloWorld:0+50
15/01/22 17:16:17 INFO PythonRDD: Times: total = 40, boot = -24261, init =
24301, finish = 0
15/01/22 17:16:17 INFO Executor: Finished task 0.0 in stage 1.0 (TID 2). 1814
bytes result sent to driver
15/01/22 17:16:17 INFO TaskSetManager: Finished task 0.0 in stage 1.0 (TID 2)
in 50 ms on localhost (1/1)
15/01/22 17:16:17 INFO TaskSchedulerImpl: Removed TaskSet 1.0, whose tasks
have all completed, from pool
15/01/22 17:16:17 INFO DAGScheduler: Stage 1 (runJob at PythonRDD.scala:344)
finished in 0.051 s
15/01/22 17:16:17 INFO DAGScheduler: Job 1 finished: runJob at PythonRDD.
scala:344, took 0.061012 s
u'Hello World!'
```

小可：嗯，程序输出结果的最后一行显示了 **Hello World！**。对照我之前输入的文件来看，这的确是文件的第一行。

Mr. 王：现在可以尝试用它来统计行数了。

```
>>> textFile.count()
15/01/22 17:15:52 WARN NativeCodeLoader: Unable to load native-hadoop library
for your platform... using builtin-java classes where applicable
15/01/22 17:15:52 WARN LoadSnappy: Snappy native library not loaded
15/01/22 17:15:52 INFO FileInputFormat: Total input paths to process : 1
15/01/22 17:15:52 INFO SparkContext: Starting job: count at <stdin>:1
15/01/22 17:15:52 INFO DAGScheduler: Got job 0 (count at <stdin>:1) with 2
output partitions (allowLocal=false)
15/01/22 17:15:52 INFO DAGScheduler: Final stage: Stage 0(count at <stdin>:1)
```

```
15/01/22 17:15:52 INFO DAGScheduler: Parents of final stage: List()
15/01/22 17:15:52 INFO DAGScheduler: Missing parents: List()
15/01/22 17:15:52 INFO DAGScheduler: Submitting Stage 0 (PythonRDD[2] at
count at <stdin>:1), which has no missing parents
15/01/22 17:15:52 INFO MemoryStore: ensureFreeSpace(5400) called with
curMem=33846, maxMem=277877882
15/01/22 17:15:52 INFO MemoryStore: Block broadcast_1 stored as values in
memory (estimated size 5.3 KB, free 265.0 MB)
15/01/22 17:15:52 INFO MemoryStore: ensureFreeSpace(3968) called with
curMem=39246, maxMem=277877882
15/01/22 17:15:52 INFO MemoryStore: Block broadcast_1_piece0 stored as bytes
in memory (estimated size 3.9 KB, free 265.0 MB)
15/01/22 17:15:52 INFO BlockManagerInfo: Added broadcast_1_piece0 in memory
on localhost:54695 (size: 3.9 KB, free: 265.0 MB)
15/01/22 17:15:52 INFO BlockManagerMaster: Updated info of block broadcast_1_
piece0
15/01/22 17:15:52 INFO SparkContext: Created broadcast 1 from broadcast at
DAGScheduler.scala:838
15/01/22 17:15:52 INFO DAGScheduler: Submitting 2 missing tasks from Stage 0
(PythonRDD[2] at count at <stdin>:1)
15/01/22 17:15:52 INFO TaskSchedulerImpl: Adding task set 0.0 with 2 tasks
15/01/22 17:15:52 INFO TaskSetManager: Starting task 0.0 in stage 0.0 (TID 0,
localhost, PROCESS_LOCAL, 1313 bytes)
15/01/22 17:15:52 INFO TaskSetManager: Starting task 1.0 in stage 0.0 (TID 1,
localhost, PROCESS_LOCAL, 1313 bytes)
15/01/22 17:15:52 INFO Executor: Running task 0.0 in stage 0.0 (TID 0)
15/01/22 17:15:52 INFO Executor: Running task 1.0 in stage 0.0 (TID 1)
15/01/22 17:15:52 INFO HadoopRDD: Input split: file:/home/lk/spark-1.2.0-bin-
hadoop1/HelloWorld:50+51
15/01/22 17:15:52 INFO HadoopRDD: Input split: file:/home/lk/spark-1.2.0-bin-
hadoop1/HelloWorld:0+50
15/01/22 17:15:52 INFO PythonRDD: Times: total = 524, boot = 435, init = 88,
finish = 1
15/01/22 17:15:52 INFO PythonRDD: Times: total = 524, boot = 431, init = 92,
finish = 1
15/01/22 17:15:52 INFO Executor: Finished task 0.0 in stage 0.0 (TID 0). 1797
bytes result sent to driver
15/01/22 17:15:52 INFO Executor: Finished task 1.0 in stage 0.0 (TID 1). 1797
bytes result sent to driver
15/01/22 17:15:52 INFO TaskSetManager: Finished task 0.0 in stage 0.0 (TID 0)
in 640 ms on localhost (1/2)
15/01/22 17:15:52 INFO TaskSetManager: Finished task 1.0 in stage 0.0 (TID 1)
in 637 ms on localhost (2/2)
15/01/22 17:15:52 INFO TaskSchedulerImpl: Removed TaskSet 0.0, whose tasks
```

```
have all completed, from pool
    15/01/22 17:15:52 INFO DAGScheduler: Stage 0 (count at <stdin>:1) finished in
0.659 s
    15/01/22 17:15:52 INFO DAGScheduler: Job 0 finished: count at <stdin>:1, took
0.796462 s
    4
```

小可：最后显示出了正确的结果！在一些运行情况信息后面，显示了一个 4，这个 4 就是行数的统计结果吧？也就是说，HelloWorld 文件有 4 行，这和我之前输入的文件是相符的。

Mr. 王：下面可以执行最后一步了，使用 filter 和 count 函数来实现最后的功能。使用命令：

```
>>> textFile.filter(lambda line: "Spark" in line).count()
```

程序的执行结果如下：

```
    15/01/22 17:17:11 INFO SparkContext: Starting job: count at <stdin>:1
    15/01/22 17:17:11 INFO DAGScheduler: Got job 2 (count at <stdin>:1) with 2
output partitions (allowLocal=false)
    15/01/22 17:17:11 INFO DAGScheduler: Final stage: Stage 2(count at <stdin>:1)
    15/01/22 17:17:11 INFO DAGScheduler: Parents of final stage: List()
    15/01/22 17:17:11 INFO DAGScheduler: Missing parents: List()
    15/01/22 17:17:11 INFO DAGScheduler: Submitting Stage 2 (PythonRDD[4] at
count at <stdin>:1), which has no missing parents
    15/01/22 17:17:11 INFO MemoryStore: ensureFreeSpace(5784) called with
curMem=51182, maxMem=277877882
    15/01/22 17:17:11 INFO MemoryStore: Block broadcast_3 stored as values in
memory (estimated size 5.6 KB, free 265.0 MB)
    15/01/22 17:17:11 INFO MemoryStore: ensureFreeSpace(4244) called with
curMem=56966, maxMem=277877882
    15/01/22 17:17:11 INFO MemoryStore: Block broadcast_3_piece0 stored as bytes
in memory (estimated size 4.1 KB, free 264.9 MB)
    15/01/22 17:17:11 INFO BlockManagerInfo: Added broadcast_3_piece0 in memory
on localhost:54695 (size: 4.1 KB, free: 265.0 MB)
    15/01/22 17:17:11 INFO BlockManagerMaster: Updated info of block broadcast_3_
piece0
    15/01/22 17:17:11 INFO SparkContext: Created broadcast 3 from broadcast at
DAGScheduler.scala:838
    15/01/22 17:17:11 INFO DAGScheduler: Submitting 2 missing tasks from Stage 2
(PythonRDD[4] at count at <stdin>:1)
    15/01/22 17:17:11 INFO TaskSchedulerImpl: Adding task set 2.0 with 2 tasks
    15/01/22 17:17:11 INFO TaskSetManager: Starting task 0.0 in stage 2.0 (TID 3,
localhost, PROCESS_LOCAL, 1313 bytes)
    15/01/22 17:17:11 INFO TaskSetManager: Starting task 1.0 in stage 2.0 (TID 4,
localhost, PROCESS_LOCAL, 1313 bytes)
    15/01/22 17:17:11 INFO Executor: Running task 0.0 in stage 2.0 (TID 3)
    15/01/22 17:17:11 INFO Executor: Running task 1.0 in stage 2.0 (TID 4)
    15/01/22 17:17:11 INFO HadoopRDD: Input split: file:/home/lk/spark-1.2.0-bin-
```

```
hadoop1/HelloWorld:0+50
    15/01/22 17:17:11 INFO HadoopRDD: Input split: file:/home/lk/spark-1.2.0-bin-
hadoop1/HelloWorld:50+51
    15/01/22 17:17:11 INFO PythonRDD: Times: total = 6, boot = 2, init = 4, finish = 0
    15/01/22 17:17:11 INFO PythonRDD: Times: total = 5, boot = 2, init = 3, finish = 0
    15/01/22 17:17:11 INFO Executor: Finished task 0.0 in stage 2.0 (TID 3). 1797
bytes result sent to driver
    15/01/22 17:17:11 INFO Executor: Finished task 1.0 in stage 2.0 (TID 4). 1797
bytes result sent to driver
    15/01/22 17:17:11 INFO TaskSetManager: Finished task 0.0 in stage 2.0 (TID 3)
in 17 ms on localhost (1/2)
    15/01/22 17:17:11 INFO TaskSetManager: Finished task 1.0 in stage 2.0 (TID 4)
in 18 ms on localhost (2/2)
    15/01/22 17:17:11 INFO TaskSchedulerImpl: Removed TaskSet 2.0, whose tasks
have all completed, from pool
    15/01/22 17:17:11 INFO DAGScheduler: Stage 2 (count at <stdin>:1) finished in
0.019 s
    15/01/22 17:17:11 INFO DAGScheduler: Job 2 finished: count at <stdin>:1, took
0.032985 s
    2
```

小可：最后这个 2 表示的就是出现过 Spark 的行数有两行吧?

小可对照了一下前面写过的 HelloWorld 文件。

小可：没错，结果是对的！的确有两行出现过 Spark 这个词！

Mr. 王：好了，我们想要实现的一个简单功能完成了。执行到这里，可以在单机上运行的 Spark 平台就已经搭建好了。不难比较出，我们使用 Spark 的单机模式基本上没有进行过配置，而且实现一些基本的文本处理功能是几乎不需要任何程序设计的，只要简单地使用一些命令或者只有一行的程序，就可以完成我们在 Hadoop 中需要几十行代码才能实现的功能，体现了它的使用是非常的简便容易的。

小可：是啊，实现这个功能只用了 3 ～ 5 行代码，的确非常的方便啊。

Mr. 王：我们休息一下，退出 Spark-Shell。

小可疑惑不解地说：咦? “Ctrl+C”快捷键为什么不好使了?

Mr. 王：哦，Python 的 PySparkShell 的快捷键不太一样，要使用“Ctrl+D”快捷键关闭它。关闭之后，Spark 还会停止一些内存和块的管理程序，程序会输出一些信息：

```
    15/01/22 17:17:20 INFO SparkUI: Stopped Spark web UI at
http://192.168.1.102:4040
    15/01/22 17:17:20 INFO DAGScheduler: Stopping DAGScheduler
    15/01/22 17:17:21 INFO MapOutputTrackerMasterActor: MapOutputTrackerActor
stopped!
    15/01/22 17:17:21 INFO MemoryStore: MemoryStore cleared
    15/01/22 17:17:21 INFO BlockManager: BlockManager stopped
```

```
15/01/22 17:17:21 INFO BlockManagerMaster: BlockManagerMaster stopped
15/01/22 17:17:21 INFO SparkContext: Successfully stopped SparkContext
15/01/22 17:17:21 INFO RemoteActorRefProvider$RemotingTerminator: Shutting
down remote daemon.
15/01/22 17:17:21 INFO RemoteActorRefProvider$RemotingTerminator: Remote
daemon shut down; proceeding with flushing remote transports.
15/01/22 17:17:21 INFO RemoteActorRefProvider$RemotingTerminator: Remoting
shut down.
lk@LKPC:~/spark-1.2.0-bin-hadoop1$
```

如果重新出现了 Shell 提示符，则说明我们已经成功地退出了 Spark。

12.2.3 在 Spark 上实现 WordCount

小可：我记得在学习 Hadoop 时，最基本的一个应用就是 WordCount，我们是不是可以用 Spark 来实现 WordCount 呢?

Mr. 王：当然可以，而且 Spark 版本的 WordCount 比在 Hadoop 下实现更加轻松、容易。

如果在 Python Spark Shell 中使用的话，则输入如下几行代码：

```
file = spark.textFile("[Filename source]")
counts = file.flatMap(lambda line: line.split(" ")) \
             .map(lambda word: (word, 1)) \
             .reduceByKey(lambda a, b: a + b)
counts.saveAsTextFile("[Filename result]")
```

这段代码就像英文的句子一样好理解吧。第一行，将输入的文件名放在引号中间，让 Spark 来识别输入的文件。第二行，定义一个变量 counts，让它等于后面的 MapReduce 的结果，后面我们将文件的每一行以空格为界限划分为单词。第三行，相当于 MapReduce 中的 Map 函数，让 Map 函数每遇到一个单词时，都将其变换成 (word,1) 这样的 key-value 对。第四行，对相当于 WordCount 中 Reduce 功能的一个定义，它会对所收到的键值相同的记录进行合并归约，对相同的 key 根据后面的格式进行变换，也就是将相同的单词所携带的计数加起来合成新的计数。最后一行，将 counts 这个结果输出到 saveAsTextFile 后面的文件中。

小可：这里有一个符号 lambda，这是什么意思呢?

Mr. 王：Spark 的基本操作是通过数据单元的变换来完成的，而这个 lambda 是用来标识变换函数的，如何执行变换也跟前面的函数名有关，后面我们还会进行详细介绍。在这里你只要记住 (lambda XXX : YYY) 相当于 (XXX => YYY) 就行，也就是将 XXX 这种格式变换成 YYY 格式。

小可恍然大悟：哦，这样就好理解多了，其实程序就是在不断地执行变换，最后将数据变换成我们所需要的格式。

Mr. 王：后面我们还会深入地讨论这个问题，现在你可以暂时这样理解。我们先来试试这

个程序吧。

小可：直接输到 Python Spark Shell 里面就可以了吗?

Mr. 王：是的。打开 Python Spark Shell，只要逐行地输入程序就可以了。

```
>>> file = sc.textFile("AnPassage")
15/01/22 20:33:05 WARN SizeEstimator: Failed to check whether
UseCompressedOops is set; assuming yes
15/01/22 20:33:05 INFO MemoryStore: ensureFreeSpace(28887) called with
curMem=0, maxMem=277877882
15/01/22 20:33:05 INFO MemoryStore: Block broadcast_0 stored as values in
memory (estimated size 28.2 KB, free 265.0 MB)
15/01/22 20:33:05 INFO MemoryStore: ensureFreeSpace(4959) called with
curMem=28887, maxMem=277877882
15/01/22 20:33:05 INFO MemoryStore: Block broadcast_0_piece0 stored as bytes
in memory (estimated size 4.8 KB, free 265.0 MB)
15/01/22 20:33:05 INFO BlockManagerInfo: Added broadcast_0_piece0 in memory
on localhost:55361 (size: 4.8 KB, free: 265.0 MB)
15/01/22 20:33:05 INFO BlockManagerMaster: Updated info of block broadcast_0_
piece0
15/01/22 20:33:05 INFO SparkContext: Created broadcast 0 from textFile at
NativeMethodAccessorImpl.java:-2
```

继续逐行地输入下面的程序，记住不要忘记后面的“\”。

```
>>> counts = file.flatMap(lambda line: line.split(" ")) \
```

有“\”的部分不会直接执行，提示符会变成 3 个点号，以等待后面的连续代码段。

```
... .map(lambda word: (word, 1)) \
... .reduceByKey(lambda a, b: a + b)
```

此时已经到达了程序定义的最后一句，不再有“\”。

```
15/01/22 20:33:41 WARN NativeCodeLoader: Unable to load native-hadoop library
for your platform... using builtin-java classes where applicable
15/01/22 20:33:41 WARN LoadSnappy: Snappy native library not loaded
15/01/22 20:33:41 INFO FileInputFormat: Total input paths to process : 1
```

最后将程序的输出路径设为 **result**，这样程序就可以开始执行了。

```
>>> counts.saveAsTextFile("result")
15/01/22 20:34:00 INFO SparkContext: Starting job: saveAsTextFile at
NativeMethodAccessorImpl.java:-2
15/01/22 20:34:00 INFO DAGScheduler: Registering RDD 4 (reduceByKey at
<stdin>:3)
15/01/22 20:34:00 INFO DAGScheduler: Got job 0 (saveAsTextFile at
NativeMethodAccessorImpl.java:-2) with 2 output partitions (allowLocal=false)
15/01/22 20:34:00 INFO DAGScheduler: Final stage: Stage 1(saveAsTextFile at
NativeMethodAccessorImpl.java:-2)
15/01/22 20:34:00 INFO DAGScheduler: Parents of final stage: List(Stage 0)
15/01/22 20:34:00 INFO DAGScheduler: Missing parents: List(Stage 0)
```

```
15/01/22 20:34:00 INFO DAGScheduler: Submitting Stage 0 (PairwiseRDD[4] at
reduceByKey at <stdin>:3), which has no missing parents
15/01/22 20:34:00 INFO MemoryStore: ensureFreeSpace(7944) called with
curMem=33846, maxMem=277877882
15/01/22 20:34:00 INFO MemoryStore: Block broadcast_1 stored as values in
memory (estimated size 7.8 KB, free 265.0 MB)
15/01/22 20:34:00 INFO MemoryStore: ensureFreeSpace(6061) called with
curMem=41790, maxMem=277877882
15/01/22 20:34:00 INFO MemoryStore: Block broadcast_1_piece0 stored as bytes
in memory (estimated size 5.9 KB, free 265.0 MB)
15/01/22 20:34:00 INFO BlockManagerInfo: Added broadcast_1_piece0 in memory
on localhost:55361 (size: 5.9 KB, free: 265.0 MB)
15/01/22 20:34:00 INFO BlockManagerMaster: Updated info of block broadcast_1_
piece0
15/01/22 20:34:00 INFO SparkContext: Created broadcast 1 from broadcast at
DAGScheduler.scala:838
15/01/22 20:34:00 INFO DAGScheduler: Submitting 2 missing tasks from Stage 0
(PairwiseRDD[4] at reduceByKey at <stdin>:3)
15/01/22 20:34:00 INFO TaskSchedulerImpl: Adding task set 0.0 with 2 tasks
15/01/22 20:34:00 INFO TaskSetManager: Starting task 0.0 in stage 0.0 (TID 0,
localhost, PROCESS_LOCAL, 1301 bytes)
15/01/22 20:34:00 INFO TaskSetManager: Starting task 1.0 in stage 0.0 (TID 1,
localhost, PROCESS_LOCAL, 1301 bytes)
15/01/22 20:34:00 INFO Executor: Running task 1.0 in stage 0.0 (TID 1)
15/01/22 20:34:00 INFO Executor: Running task 0.0 in stage 0.0 (TID 0)
15/01/22 20:34:01 INFO HadoopRDD: Input split: file:/home/lk/spark-1.2.0-bin-
hadoop1/AnPassage:0+56
15/01/22 20:34:01 INFO HadoopRDD: Input split: file:/home/lk/spark-1.2.0-bin-
hadoop1/AnPassage:56+57
15/01/22 20:34:01 INFO PythonRDD: Times: total = 505, boot = 429, init = 75,
finish = 1
15/01/22 20:34:01 INFO PythonRDD: Times: total = 505, boot = 426, init = 78,
finish = 1
15/01/22 20:34:01 INFO Executor: Finished task 0.0 in stage 0.0 (TID 0). 1958
bytes result sent to driver
15/01/22 20:34:01 INFO Executor: Finished task 1.0 in stage 0.0 (TID 1). 1958
bytes result sent to driver
15/01/22 20:34:01 INFO TaskSetManager: Finished task 1.0 in stage 0.0 (TID 1)
in 652 ms on localhost (1/2)
15/01/22 20:34:01 INFO TaskSetManager: Finished task 0.0 in stage 0.0 (TID 0)
in 663 ms on localhost (2/2)
15/01/22 20:34:01 INFO TaskSchedulerImpl: Removed TaskSet 0.0, whose tasks
have all completed, from pool
15/01/22 20:34:01 INFO DAGScheduler: Stage 0 (reduceByKey at <stdin>:3)
```

```
finished in 0.678 s
    15/01/22 20:34:01 INFO DAGScheduler: looking for newly runnable stages
    15/01/22 20:34:01 INFO DAGScheduler: running: Set()
    15/01/22 20:34:01 INFO DAGScheduler: waiting: Set(Stage 1)
    15/01/22 20:34:01 INFO DAGScheduler: failed: Set()
    15/01/22 20:34:01 INFO DAGScheduler: Missing parents for Stage 1: List()
    15/01/22 20:34:01 INFO DAGScheduler: Submitting Stage 1 (MappedRDD[9] at
saveAsTextFile at NativeMethodAccessorImpl.java:-2), which is now runnable
    15/01/22 20:34:01 INFO MemoryStore: ensureFreeSpace(20856) called with
curMem=47851, maxMem=277877882
    15/01/22 20:34:01 INFO MemoryStore: Block broadcast_2 stored as values in
memory (estimated size 20.4 KB, free 264.9 MB)
    15/01/22 20:34:01 INFO MemoryStore: ensureFreeSpace(15342) called with
curMem=68707, maxMem=277877882
    15/01/22 20:34:01 INFO MemoryStore: Block broadcast_2_piece0 stored as bytes
in memory (estimated size 15.0 KB, free 264.9 MB)
    15/01/22 20:34:01 INFO BlockManagerInfo: Added broadcast_2_piece0 in memory
on localhost:55361 (size: 15.0 KB, free: 265.0 MB)
    15/01/22 20:34:01 INFO BlockManagerMaster: Updated info of block broadcast_2_
piece0
    15/01/22 20:34:01 INFO SparkContext: Created broadcast 2 from broadcast at
DAGScheduler.scala:838
    15/01/22 20:34:01 INFO DAGScheduler: Submitting 2 missing tasks from Stage 1
(MappedRDD[9] at saveAsTextFile at NativeMethodAccessorImpl.java:-2)
    15/01/22 20:34:01 INFO TaskSchedulerImpl: Adding task set 1.0 with 2 tasks
    15/01/22 20:34:01 INFO TaskSetManager: Starting task 0.0 in stage 1.0 (TID 2,
localhost, PROCESS_LOCAL, 1056 bytes)
    15/01/22 20:34:01 INFO TaskSetManager: Starting task 1.0 in stage 1.0 (TID 3,
localhost, PROCESS_LOCAL, 1056 bytes)
    15/01/22 20:34:01 INFO Executor: Running task 0.0 in stage 1.0 (TID 2)
    15/01/22 20:34:01 INFO Executor: Running task 1.0 in stage 1.0 (TID 3)
    15/01/22 20:34:01 INFO ShuffleBlockFetcherIterator: Getting 2 non-empty blocks
out of 2 blocks
    15/01/22 20:34:01 INFO ShuffleBlockFetcherIterator: Getting 2 non-empty blocks
out of 2 blocks
    15/01/22 20:34:01 INFO ShuffleBlockFetcherIterator: Started 0 remote fetches
in 6 ms
    15/01/22 20:34:01 INFO ShuffleBlockFetcherIterator: Started 0 remote fetches
in 6 ms
    15/01/22 20:34:01 INFO PythonRDD: Times: total = 69, boot = -152, init = 221,
finish = 0
    15/01/22 20:34:01 INFO PythonRDD: Times: total = 74, boot = -147, init = 221,
finish = 0
    15/01/22 20:34:01 INFO FileOutputCommitter: Saved output of task
```

```
'attempt_201501222034_0001_m_000000_2' to file:/home/lk/spark-1.2.0-bin-hadoop1/
result
    15/01/22 20:34:01 INFO FileOutputCommitter: Saved output of task
'attempt_201501222034_0001_m_000001_3' to file:/home/lk/spark-1.2.0-bin-hadoop1/
result
    15/01/22 20:34:01 INFO SparkHadoopWriter: attempt_201501222034_0001_
m_000000_2: Committed
    15/01/22 20:34:01 INFO SparkHadoopWriter: attempt_201501222034_0001_
m_000001_3: Committed
    15/01/22 20:34:01 INFO Executor: Finished task 0.0 in stage 1.0 (TID 2). 886
bytes result sent to driver
    15/01/22 20:34:01 INFO Executor: Finished task 1.0 in stage 1.0 (TID 3). 886
bytes result sent to driver
    15/01/22 20:34:01 INFO TaskSetManager: Finished task 0.0 in stage 1.0 (TID 2)
in 119 ms on localhost (1/2)
    15/01/22 20:34:01 INFO TaskSetManager: Finished task 1.0 in stage 1.0 (TID 3)
in 119 ms on localhost (2/2)
    15/01/22 20:34:01 INFO TaskSchedulerImpl: Removed TaskSet 1.0, whose tasks
have all completed, from pool
    15/01/22 20:34:01 INFO DAGScheduler: Stage 1 (saveAsTextFile at
NativeMethodAccessorImpl.java:-2) finished in 0.120 s
    15/01/22 20:34:01 INFO DAGScheduler: Job 0 finished: saveAsTextFile at
NativeMethodAccessorImpl.java:-2, took 0.950491 s
    >>>
```

程序执行之后，平台会自动结束。

```
    15/01/22 20:34:06 INFO SparkUI: Stopped Spark web UI at http://LKPC.
local:4040
    15/01/22 20:34:06 INFO DAGScheduler: Stopping DAGScheduler
    15/01/22 20:34:07 INFO MapOutputTrackerMasterActor: MapOutputTrackerActor
stopped!
    15/01/22 20:34:07 INFO MemoryStore: MemoryStore cleared
    15/01/22 20:34:07 INFO BlockManager: BlockManager stopped
    15/01/22 20:34:07 INFO BlockManagerMaster: BlockManagerMaster stopped
    15/01/22 20:34:07 INFO SparkContext: Successfully stopped SparkContext
    15/01/22 20:34:07 INFO RemoteActorRefProvider$RemotingTerminator: Shutting
down remote daemon.
    15/01/22 20:34:07 INFO RemoteActorRefProvider$RemotingTerminator: Remote
daemon shut down; proceeding with flushing remote transports.
    15/01/22 20:34:07 INFO RemoteActorRefProvider$RemotingTerminator: Remoting
shut down.
```

这时我们可以在目录下查看文件列表，使用 ls 命令：

```
    lk@LKPC:~/spark-1.2.0-bin-hadoop1$ ls
```

我们会发现文件列表中多出了一个 result 文件夹，这个文件夹就是前面程序的输出结果。我们可以使用 cd 命令进入文件夹，并且继续使用 ls 命令查看里面的文件。

```
lk@LKPC:~/spark-1.2.0-bin-hadoop1$ cd result/
lk@LKPC:~/spark-1.2.0-bin-hadoop1/result$ ls
```

小可：和 Hadoop 很像，里面会出现 Part-00000 和 Part-00001 这样的文件。

```
part-00000  part-00001  _SUCCESS
```

Mr. 王：打开看看，里面就保存着 WordCount 的结果。

12.2.4　在 HDFS 上使用 Spark

Mr. 王：前面我们使用 Spark 平台时，输入文件都来自于本地磁盘，为了能够更加方便地在多机器上使用 Spark，我们可以将输入输出文件存放在分布式文件系统上。

小可：分布式文件系统……在学习 Hadoop 时，我们使用的是 HDFS，这次我们还用 HDFS 可以吗?

Mr. 王：很好，Spark 依然可以将输入输出文件放在 HDFS 上，以便于在多台计算机上运行 Spark 程序。这次，输入文件将不再来自于本地磁盘，而是来自于 HDFS。

首先，我们要启动 HDFS 服务。

```
lk@LKPC:~/hadoop-1.0.1$ bin/start-all.sh
```

然后，还要记得格式化 HDFS。

```
lk@LKPC:~/hadoop-1.0.1$ bin/hadoop namenode -format
```

最后，我们去 HDFS 的 Web 页面检查一下 HDFS 是不是已经正常启动并格式化好了。用浏览器打开 URL：

```
localhost:50070
```

小可：嗯，又看到熟悉的画面了，HDFS 的 NameNode 状态信息显示出来了，HDFS 已经可以正常启动了。

Mr. 王：现在我们在本地创建一个包含一些随机句子的文本文件。

实验使用的文本文件的内容如下：

```
I am a rookie of Spark
I am studying about big data now
I know a bit about big data now
Spark is a good platform
It helps a lot
Thanks
I like it
Have fun with Spark
Good luck
```

然后将它放入 HDFS 中，使用 HDFS 的 -put 命令，依然要注意放置文件的路径关系。

```
lk@LKPC:~/hadoop-1.0.1$ bin/hadoop dfs -put  ./inputfile  /user/lk/inputfile
```

再用 ls 命令查看一下，文件是不是已经成功地放进去了。

```
lk@LKPC:~/hadoop-1.0.1$ bin/hadoop dfs -ls
Found 1 items
-rw-r--r--   1 lk supergroup        175 2015-01-23 17:20 /user/lk/inputfile
```

小可：找到了，这就是我们刚刚放进去的文本文件！

Mr. 王：好的，接下来可以去 Spark 那里，执行下一步工作了。

使用切换目录的命令：

cd ..（注意不要丢掉中间的空格）

cd spark-1.2.0-bin-hadoop1/（Spark 解压缩路径，不同版本不一样）

Mr. 王：接下来还是一样启动 Python Spark Shell。

```
lk@LKPC:~/spark-1.2.0-bin-hadoop1$ bin/pyspark
```

在大段提示信息之后，出现“>>>”命令提示符。

我们依然采用下面的格式来输入文本文件。

```
file = sc.textFile("hdfs://localhost:9000/user/lk/inputfile")
```

这里注意，输入文件如果来自于 HDFS，则要在文件路径前面加 hdfs://，以便系统识别。后面部分是 HDFS 的访问路径，由于本实验是在本地进行的，所以输入 localhost:9000，这是访问 HDFS 的路径。接下来就是文件放置的位置，和前面放进 HDFS 时是一致的。

在完成了从 HDFS 加载文件之后，我们就可以按照需要完成接下来的操作了。我们要做的是选出所有句子中带有“Spark”关键词的句子，并将它们返回。

```
>>> spks = file.filter(lambda line: "Spark" in line)
>>> spks.filter(lambda line: "Spark" in line).collect()
```

程序的输出结果如下：

```
    15/01/23 17:23:54 WARN NativeCodeLoader: Unable to load native-hadoop library
for your platform... using builtin-java classes where applicable
    15/01/23 17:23:54 WARN LoadSnappy: Snappy native library not loaded
    15/01/23 17:23:54 INFO FileInputFormat: Total input paths to process : 1
    15/01/23 17:23:54 INFO SparkContext: Starting job: collect at <stdin>:1
    15/01/23 17:23:54 INFO DAGScheduler: Got job 0 (collect at <stdin>:1) with 2
output partitions (allowLocal=false)
    15/01/23 17:23:54 INFO DAGScheduler: Final stage: Stage 0(collect at
<stdin>:1)
    15/01/23 17:23:54 INFO DAGScheduler: Parents of final stage: List()
    15/01/23 17:23:54 INFO DAGScheduler: Missing parents: List()
    15/01/23 17:23:54 INFO DAGScheduler: Submitting Stage 0 (PythonRDD[2] at
collect at <stdin>:1), which has no missing parents
    15/01/23 17:23:54 INFO MemoryStore: ensureFreeSpace(5032) called with
curMem=33846, maxMem=277877882
    15/01/23 17:23:54 INFO MemoryStore: Block broadcast_1 stored as values in
memory (estimated size 4.9 KB, free 265.0 MB)
```

```
15/01/23 17:23:54 INFO MemoryStore: ensureFreeSpace(3712) called with
curMem=38878, maxMem=277877882
15/01/23 17:23:54 INFO MemoryStore: Block broadcast_1_piece0 stored as bytes
in memory (estimated size 3.6 KB, free 265.0 MB)
15/01/23 17:23:54 INFO BlockManagerInfo: Added broadcast_1_piece0 in memory
on localhost:40510 (size: 3.6 KB, free: 265.0 MB)
15/01/23 17:23:54 INFO BlockManagerMaster: Updated info of block broadcast_1_
piece0
15/01/23 17:23:54 INFO SparkContext: Created broadcast 1 from broadcast at
DAGScheduler.scala:838
15/01/23 17:23:54 INFO DAGScheduler: Submitting 2 missing tasks from Stage 0
(PythonRDD[2] at collect at <stdin>:1)
15/01/23 17:23:54 INFO TaskSchedulerImpl: Adding task set 0.0 with 2 tasks
15/01/23 17:23:54 INFO TaskSetManager: Starting task 0.0 in stage 0.0 (TID 0,
localhost, ANY, 1304 bytes)
15/01/23 17:23:54 INFO TaskSetManager: Starting task 1.0 in stage 0.0 (TID 1,
localhost, ANY, 1304 bytes)
15/01/23 17:23:54 INFO Executor: Running task 0.0 in stage 0.0 (TID 0)
15/01/23 17:23:54 INFO Executor: Running task 1.0 in stage 0.0 (TID 1)
15/01/23 17:23:55 INFO HadoopRDD: Input split: hdfs://localhost:9000/user/lk/
inputfile:0+87
15/01/23 17:23:55 INFO HadoopRDD: Input split: hdfs://localhost:9000/user/lk/
inputfile:87+88
15/01/23 17:23:55 INFO PythonRDD: Times: total = 553, boot = 442, init = 110,
finish = 1
15/01/23 17:23:55 INFO PythonRDD: Times: total = 553, boot = 446, init = 106,
finish = 1
15/01/23 17:23:55 INFO Executor: Finished task 0.0 in stage 0.0 (TID 0). 1824
bytes result sent to driver
15/01/23 17:23:55 INFO Executor: Finished task 1.0 in stage 0.0 (TID 1). 1869
bytes result sent to driver
15/01/23 17:23:55 INFO TaskSetManager: Finished task 0.0 in stage 0.0 (TID 0)
in 699 ms on localhost (1/2)
15/01/23 17:23:55 INFO TaskSetManager: Finished task 1.0 in stage 0.0 (TID 1)
in 693 ms on localhost (2/2)
15/01/23 17:23:55 INFO TaskSchedulerImpl: Removed TaskSet 0.0, whose tasks
have all completed, from pool
15/01/23 17:23:55 INFO DAGScheduler: Stage 0 (collect at <stdin>:1) finished
in 0.715 s
15/01/23 17:23:55 INFO DAGScheduler: Job 0 finished: collect at <stdin>:1,
took 0.827310 s
[u'I am a rookie of Spark', u'Spark is a good platform', u'Have fun with
Spark']
```

注意观察输出结果的最后一行，三个引号里面的部分分别是 I am a rookie of Spark、Spark is a good platform、Have fun with Spark 三个句子。对比一下之前我们输入的文件，这的确是 Spark 出现的三个地方，运行结果还是比较准确的。

另外，还记得我们在学习 WordCount 时用过的 saveAsTextFile 函数吗?

我们同样可以使用下面这条命令，将运行结果存储到 HDFS 中，这样更加符合分布式并行计算产生结果的数据量同样比较大这个特点。

```
>>>spks.saveAsTextFile("hdfs://localost:9000/user/lk/result")
```

12.2.5 Spark 的核心操作——Transformation 和 Action

Mr. 王：通过前面三个简单的小程序，相信你已经对 Spark 的使用有了一个初步的认识。不过仅仅这样还不够，还要了解分析 Spark 操作的核心内容，我们要知道，Spark 是依照什么样的规则去操作数据的。

我们先来看看前面写过的两行代码：

```
>>> spks = file.filter(lambda line: "Spark" in line)
>>> spks.filter(lambda line: "Spark" in line).collect()
```

这两行代码虽然非常简单，但却体现了 Spark 最核心的两个基本操作，即 Transformation 和 Action。

小可：Hadoop 有 Map 和 Reduce，Spark 有 Transformation 和 Action，挺有意思的。

Mr. 王：嗯，但它们并不完全类似。在学习 Spark 的过程中，除了要记住它处理数据和保存中间结果的方式是 RDD，而不是 Hadoop 面向磁盘的 HDFS 之外。最重要的就是要了解这两个关键操作：Transformation 和 Action。

小可：那什么是 Transformation 和 Action 呢?

Mr. 王：顾名思义，Transformation 就是变换，它的作用是将已有的 RDD 转换成新的 RDD。

这是提出 Spark 平台的论文中给出的 Transformation 的各种操作汇总表格。其中就包括我们之前使用的 filter。表格中给出的 Transformation 操作其实都非常好理解，就拿 filter 操作来说：

```
filter( f: T => Bool )  :  RDD[T] => RDD [T]
```

其中，输入 T 就是原始的数据集合，filter 根据映射关系 f，将原始的数据集合 T 构成的 RDD 转化成一个个新的集合 RDD[T]，里面的数据都来自于原来的数据集合，但它必须满足某条件，根据其布尔类型结果来判断它是不是应该被加入到变换之后的 RDD[T] 中。我们知道，布尔类型就是真和假，如果满足某条件，我们称之为真，反之称之为假。就拿我们的例子来说，如果某一行数据中包含“Spark”关键词的话，映射关系 f 就会将其确定为真，否则为假。从本质上来讲，filter 相当于进行了一个条件筛选工作。

Spark 支持的对 RDD（弹性分布式数据集合）的 Transformation（变换）和 Action（操作），其中 Seq[T] 表示的是类型为 T 的元素组成的序列

Transformations	*map*(*f* : T ⇒ U)	:	RDD[T] ⇒ RDD[U]
	filter(*f* : T ⇒ Bool)	:	RDD[T] ⇒ RDD[T]
	flatMap(*f* : T ⇒ Seq[U])	:	RDD[T] ⇒ RDD[U]
	sample(*fraction* : Float)	:	RDD[T] ⇒ RDD[T] (Deterministic sampling)
	groupByKey()	:	RDD[(K, V)] ⇒ RDD[(K, Seq[V])]
	reduceByKey(*f* : (V, V) ⇒ V)	:	RDD[(K, V)] ⇒ RDD[(K, V)]
	union()	:	(RDD[T], RDD[T]) ⇒ RDD[T]
	join()	:	(RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (V, W))]
	cogroup()	:	(RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (Seq[V], Seq[W]))]
	crossProduct()	:	(RDD[T], RDD[U]) ⇒ RDD[(T, U)]
	mapValues(*f* : V ⇒ W)	:	RDD[(K, V)] ⇒ RDD[(K, W)] (Preserves partitioning)
	sort(*c* : Comparator[K])	:	RDD[(K, V)] ⇒ RDD[(K, V)]
	partitionBy(*p* : Partitioner[K])	:	RDD[(K, V)] ⇒ RDD[(K, V)]
Actions	*count*()	:	RDD[T] ⇒ Long
	collect()	:	RDD[T] ⇒ Seq[T]
	reduce(*f* : (T, T) ⇒ T)	:	RDD[T] ⇒ T
	lookup(*k* : K)	:	RDD[(K, V)] ⇒ Seq[V] (On hash/range partitioned RDDs)
	save(*path* : String)	:	Outputs RDD to a storage system, *e.g.*, HDFS

小可：哦！这样就实现了将含有“Spark”关键词的句子都挑选出来的功能，它们都会被映射关系 *f* 标记为真。

Mr. 王：是这样的。仔细观察会发现，Transformation 中定义的操作都非常的实用，可以满足我们对数据集合的各种变换操作。

比如 Map：

```
map ( f: T => U )  :  RDD[T] => RDD [U]
```

将一个数据集合映射变换为另一个数据集合，适合我们进行各种变换操作。

这个操作我们在编写 WordCount 时是使用过的。那时，要将句子切分好的单词 word 映射变换成 (word,1) 这样的键值对。

小可：此处的 map 函数就起到了在 Hadoop 版本的 WordCount 中 Mapper 的作用，将文档中的单词变换为 (word,1) 这样的键值对。

Mr. 王：另外一个非常常用的函数就是 reduceByKey：

```
reduceByKey( f(V,V) => V) : RDD[K,V] => RDD[K,V]
```

小可：这个函数我们在 WordCount 中也使用过。

Mr. 王：顾名思义，reduceByKey 会对具有相同键值的 key-value 对执行一个操作，这个操作由函数 *f* 进行定义。

小可：哦，观察 WordCount 中的 reduceByKey 的确可以发现：

```
.reduceByKey(lambda a, b: a + b)
```

就是对相同键值后面携带的两个值 *a* 和 *b* 求 *a*+*b* 之后，变成两个键值对合并之后的新 value。

Mr. 王：第一轮变换，将所有的关键词都映射成 (word，1) 这样的键值对，比如 (Hello,1)。在这一步执行 reduceByKey 时，两个 (Hello,1) 相遇，就会被执行 (a,b:a+b) 操作，也就是合并

为 (Hello,2)。相应的操作会继续执行下去，直到所有的 (Hello,1) 都完成了合并，得到 Hello 关键词的最终计数结果。

小可：在 WordCount 中，reduceByKey 执行了相当于 Hadoop 中 Reduce 的操作。

Mr. 王：groupByKey 算是 reduceByKey 的一个特殊情况，它执行的就是将具有相同 key 值的键值对进行合并，使这些键值对的 value 构成一个列表，并以 key 值和这个列表组成新的键值对，构成 RDD。

小可：这也是一个很实用的功能啊。

Mr. 王：像 union、join、sort、crossProduct 这样的操作从名字上就非常容易理解，它们可以实现合并、值组合连接、排序、叉积这些非常常用的操作，也为基于 Spark 实现各种数据库操作、实现基于 Spark 的 SQL（数据库查询语言）语句提供了重要的基础。可以看出，Spark 为我们提供的 API 还是非常强大的。

小可：Action 又是什么呢?

Mr. 王：事实上，当向 Spark 平台写入 Transformation 时，Spark 并不会立即执行 Transformation 操作，它更多的是对来自于 RDD 的数据变换形式进行定义，当 Action 操作被输入到 Spark 中时，才会真正地开始进行实际的运算。Spark 会根据前面定义的数据变换形式和 Action 执行的具体操作，将需要各种工作真正地分配给机群去执行。

我们来看看 Action 里面包含的操作。

```
count() : RDD[T] => Long
```

这是一个非常典型的 Action 操作，就是对数据集合 RDD 或者是经过多轮 Transformation 变换的那些 RDD 的记录个数进行统计的操作。我们执行的第一个 Spark 程序行计数就使用了这个操作。

另一个非常常用的操作就是 collect。

小可：我们在查找某个关键词出现的所有句子时，就使用了这个操作。

Mr. 王：我们来看看它的定义。

```
collect : RDD[T] => Seq [T]
```

其实这个操作也很简单，它将 RDD 中所有的数据记录收集起来，形成一个列表，以便于之后的保存等操作。这个操作往往要配合前面的各种变换进行，用于生成结果列表。

其实我们还使用过 save 这个操作，它可以将一个 RDD 存储为文件，一般用来存储大量的处理结果，可以存储在像 HDFS 这样的文件系统中。

小可：就是我们使用过的 saveAsTextFile 函数吧。

Mr. 王：是的。了解了 Transformation 和 Action 这两种操作，就可以用 Spark 设计绝大多数的并行程序了，回去多多尝试，很快就可以熟练地运用它们，这也是并行编程框架为我们提供的方便。

12.2.6　Spark 实践案例——PageRank

Mr. 王：了解了 Spark 的基础使用和两种基本操作之后，我们来尝试实现更加有实际应用意义的案例——PageRank。PageRank 我们前面提到过，是谷歌提出的著名的网页重要度评价算法。其实这个算法并不是很复杂，但我们要用并行平台 Spark 来实现有两个原因。

第一，PageRank 算法虽然简单，但是网络中的网页数目却非常巨大，而且网页页面也是非常复杂的，有着众多的链接，可以想象用来表示一个实际网络的连接关系的数据量将会非常大，这意味着处理它们也会变得有一定的难度。为了让其效率变得更高，我们就需要引入多台计算机来进行处理，也就是进行并行计算。

第二，PageRank 是一个通过不断研究多级链接关系去更新每个网页重要程度的算法，这意味着它会经历多轮迭代；而 Spark 的 in-memory 计算的思想恰恰非常适合迭代运算，所以我们选用 Spark。

小可：那么具体该怎么做呢？

Mr. 王：还是先回忆一下这个算法的输入输出格式。

小可：PageRank 最重要的输入数据就是表示网页之间的链接关系，所以输入只要包括网页的链接关系就好了。用一个文本文件，将每一个链接表示为：

```
链接源 链接目的
```

这样的形式。比如，如果 www.1.com 向 www.2.cn 有一个链接的话，我们就在文本文件中记录下：

```
www.1.com  www.2.cm
```

Mr. 王：嗯，那输出呢？

小可：输出就是我们期待的结果，将各个网站的重要程度分数输出出来即可。

Mr. 王：很好，我们先来看看这个程序。

```
import re
import sys
from operator import add

from pyspark import SparkContext


def computeContribs(urls, rank):
    num_urls = len(urls)
    for url in urls:
        yield (url, rank / num_urls)


def parseNeighbors(urls):
    parts = re.split(r'\s+', urls)
    return parts[0], parts[1]
```

```
    if __name__ == "__main__":
        if len(sys.argv) != 3:
            print >> sys.stderr, "Usage: pagerank <file> <iterations>"
            exit(-1)

        print >> sys.stderr,  """WARN: This is a naive implementation of PageRank
and is
                given as an example! Please refer to PageRank implementation
provided by graphx"""

        sc = SparkContext(appName="PythonPageRank")

        lines = sc.textFile(sys.argv[1], 1)

          links = lines.map(lambda urls: parseNeighbors(urls)).distinct().
groupByKey().cache()

        ranks = links.map(lambda (url, neighbors): (url, 1.0))
        for iteration in xrange(int(sys.argv[2])):
            contribs = links.join(ranks).flatMap(
                lambda (url, (urls, rank)): computeContribs(urls, rank))

             ranks = contribs.reduceByKey(add).mapValues(lambda rank: rank * 0.85
+ 0.15)

        for (link, rank) in ranks.collect():
            print "%s has rank: %s." % (link, rank)
        sc.stop()
```

这个程序并不长，我们对其做一个简单的分析。

```
    def computeContribs(urls, rank):
        num_urls = len(urls)
        for url in urls:
            yield (url, rank / num_urls)
    def parseNeighbors(urls):
        parts = re.split(r'\s+', urls)
        return parts[0], parts[1]
```

在程序的一开始，首先定义了两个小函数，让一些重复使用的基本操作不必写到大的程序框架中。**computeContribs** 用于计算一个网页地址对其他网页地址的贡献；**parseNeighbors** 是一个格式转换函数，用于将我们写在文本文件里的整个地址对转换成连接的组合。

接下来这几行代码是用于保障程序健壮性的。执行 **pagerank** 需要两个参数，其中一个是表

示网页连接关系的文件；另一个是迭代次数。

```
if len(sys.argv) != 3:
    print >> sys.stderr, "Usage: pagerank <file> <iterations>"
    exit(-1)
```

小可：为什么需要迭代次数呢?

Mr. 王：在一些特殊的情况下，网页的连接关系构成了一个相互增益的环形，或者是形成了很长很深的链状，导致程序中的循环会运行很多次，或者由于不断的循环增益，很多网页的连接关系会不断地更新而根本不能停止，所以我们需要设置迭代次数的最大值，以避免程序无休止地运行。

下面这部分想必你已经非常熟悉了，将前面表示各个网页连接关系的文件作为 textFile 输入到 Spark 中去。

```
lines = sc.textFile(sys.argv[1], 1)
```

注意，这里出现了第一个 map 操作。

```
links = lines.map(lambda urls: parseNeighbors(urls)).distinct().groupByKey().
cache()
```

它将网页映射成其可以处理的网页地址对，以便进行进一步的处理。后面它使用了 Spark 的 distinct() 函数进行数据去重，以防止重复的记录干扰到计算结果；groupByKey() 将具有相同键值的网页连接关系聚集起来，并且使用 cache() 将这些结果缓存起来。

接下来我们将管理好的数据记录映射成网页和 1.0 这种形式。后面的 1.0 是对每个网页重要程度的初始化，刚一开始时网页的重要程度都是 1。

```
ranks = links.map(lambda (url, neighbors): (url, 1.0))
```

现在开始进入 PageRank 的核心部分，整个程序会迭代执行，次数为我们设定的最大迭代次数。

```
for iteration in xrange(int(sys.argv[2])):
```

在每一轮迭代的过程中，首先计算每个网页对其他网页的贡献，其中使用了前面定义过的函数进行贡献计算，并将其存储在 contribs 变量中。

```
contribs = links.join(ranks).flatMap(
,     lambda (url, (urls, rank)): computeContribs(urls, rank))
```

然后根据每个网页在本轮迭代中获得的其他网页对自己的贡献程度，对每个网页更新其重要程度评分。

```
ranks = contribs.reduceByKey(add).mapValues(lambda rank: rank * 0.85 + 0.15)
```

接下来程序会执行下一轮，直到不再发生变化，或者已经达到最大迭代次数为止。

最后程序会收集所有的网页获得的重要程度评分，并输出网页的重要程度评分。

```
for (link, rank) in ranks.collect():
    print "%s has rank: %s." % (link, rank)
```

我们用下面这个例子作为输入，来试着执行程序。

文件名：links

```
www.1.com www.2.com
www.1.com www.3.com
www.2.com www.3.com
www.1.com www.4.com
www.4.com www.2.com
www.5.com www.2.com
www.6.com www.1.com
www.2.com www.6.com
www.3.com www.5.com
```

小可：这么长的程序，使用 Python Spark Shell 一句句地输入进去岂不是很麻烦吗？

Mr. 王：嗯，这是我要说明的另一个问题，就是如何让 Spark 直接执行一个 Python 脚本。这个功能是非常有必要的，当要进行的操作相对复杂一些时，我们不可能让整个程序都一句句地直接输入到 pyspark 中，这样不仅很麻烦，而且也不利于代码的重复使用。Spark 也考虑到了这一点，提供了一个直接执行脚本的方法。

```
$ bin/spark-submit [ 程序文件名 ] [ 参数 ]
```

在这里我们就可以使用：

```
bin/spark-submit examples/src/main/python pagerank.py links 10
```

屏幕上开始滚动着大量的执行过程信息。

小可：执行结果出来了，果然非常方便啊！各个网页的重要程度都显示出来了！

```
...
www.3.com has rank: 1.106284947.
www.2.com has rank: 1.6780673438.
www.4.com has rank: 0.403694401488.
www.6.com has rank: 0.85259054551.
www.1.com has rank: 0.876251438162.
www.5.com has rank: 1.08311132405.
```

Mr. 王：好了，到此为止，我们已经对 Spark 有了一个初步的认识，了解了 Spark 的两种基本操作，你可以很快地学会用它执行各种不同算法的方法了。接下来你可以去参考 Spark 的官方网站或者网络上的各种教程。Enjoy yourself with Spark！

第13章 众包算法实践

13.1 认识 AMT

Mr. 王：今天我们来讨论一下众包平台的实际使用。

小可：嗯，在前面介绍时就觉得众包是一个非常有趣的算法思想，我也很想了解它的具体使用呢。

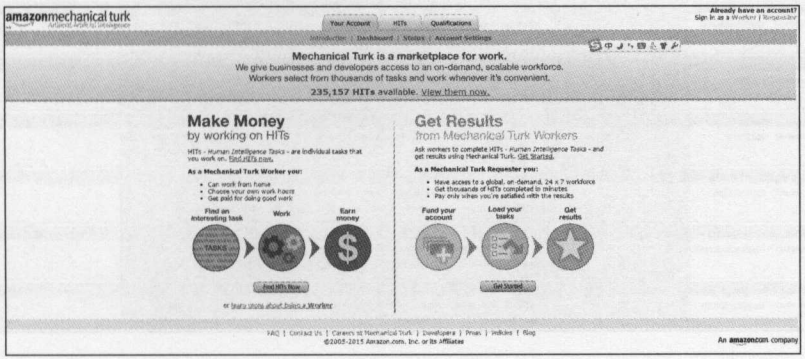
Mr. 王：现在我们就以一个具体的众包平台为例，谈谈如何使用众包平台。

大多数时候，我们见到的众包平台都是以网站为表现形式的，在这里我们就以一个非常著名的众包平台——Amazon 的 Mechanical Turk 为例，了解如何使用众包平台完成任务和发布任务。

Mechanical Turk 是 Amazon Web Service（AWS）的组成部分之一，是一个非常典型的众包平台，它的网址是 https://www.mturk.com/mturk/welcome，可以直接通过浏览器进行访问。

小可：那我们就来试试吧。

小可迫不及待地打开电脑，输入了刚才的网址。

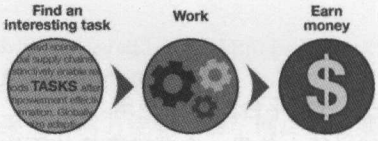


小可：是一个英文网站。

Mr. 王：没错，这就是 Amazon Mechanical Turk，一个非常典型的众包平台。从这个网站的主界面就可以非常清晰地看出它分为两个功能部分；左边写着 Make Money by working on HITs 的部分就是供 Worker 选择任务发布者提出的任务，并且完成它们获得相应回报的区域；右边的 Get Results from Mechanical Turk Workers 部分就是留给任务发布者去发布任务的区域。

13.2 成为众包工人

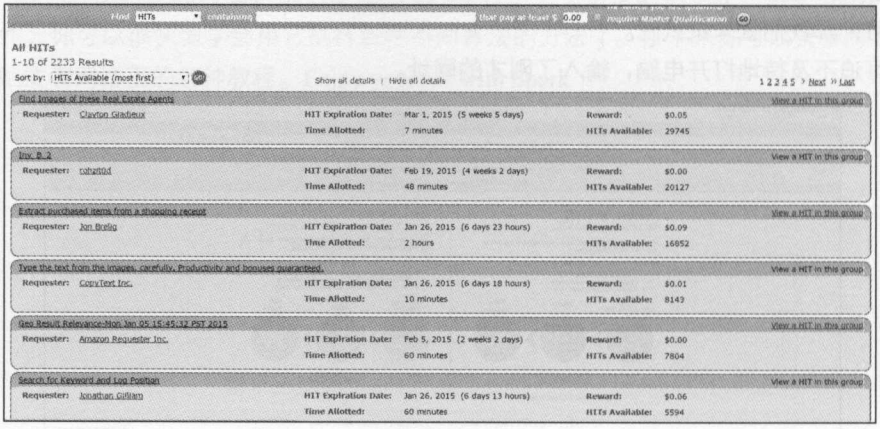
首先我们来看看作为工人去完成任务这一部分。网站使用了非常鲜明的图片来表示这一部分的使用过程。我们可以先到众多的任务中去找一个自己感兴趣的任务，然后选择工作时间，一般我们都是利用业余时间来完成众包任务，而且多数时候都是在家里完成的。在完成之后，我们可以得到任务发布者提供的报酬。在 AMT 上，一般报酬是以金钱的形式进行支付的，工人可以注册亚马逊的账户来收取报酬。



小可：网站上还说 HIT 就是作为工人可以去完成的一项工作，后面的 Find HITs now 是不是能让工人直接去寻找这些 HIT 呢？

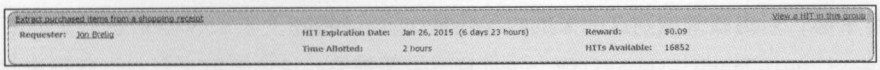
> HITs - *Human Intelligence Tasks* - are individual tasks that you work on. Find HITs now.

小可点击了链接，弹出了一个新的页面。



小可：哇，弹出了好多个项目。

Mr. 王：这些项目就是 HIT 的说明。

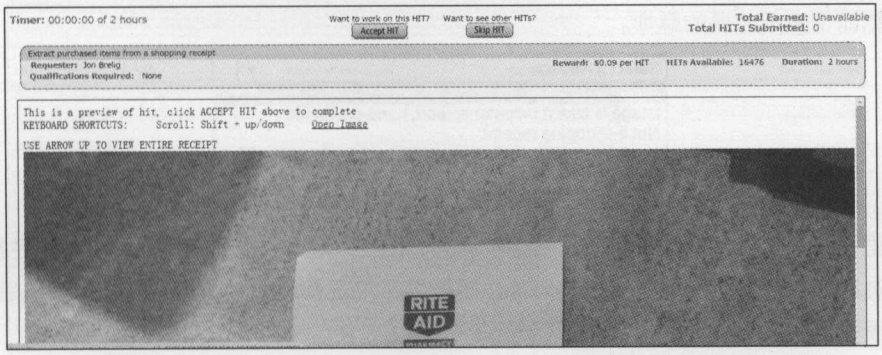


我们以其中的一个为例，简单地解释一下。从题目中不难看出，任务的提供者希望工人能从购物小票中提取出客户购买的商品。在下面我们可以看到这个任务的请求者名字、任务的截止日期等，并且还给出了完成一个 HIT 需要的时间。

小可：Reward 一定就是任务的报酬了，看来该网站以美元结算。这个任务每完成一个 HIT，就可以收到 9 美分的回报。

Mr. 王：嗯，众包任务的发布者可以以金钱形式作为回报来吸引工人投入到完成任务工作中。工人通过完成任务不仅消磨掉了自己的无聊时间，而且还有一定的收益。同时，任务的提供者也通过支付一点点报酬达到了完成任务的目的。

小可：我点击 View a HIT in this group 链接试试看。



Mr. 王：弹出的这个新页面是预览一个任务，一般用于给工人提供一个任务的例子，并且说明如何完成这个任务，并给出完成一个任务的要求等。

根据任务的标题我们可以推测出，它会给出一个购物小票的照片，这些照片往往是不太规整、字体较小或者字迹比较模糊的，总而言之，让计算机去处理这样的问题会遇到一些困难。

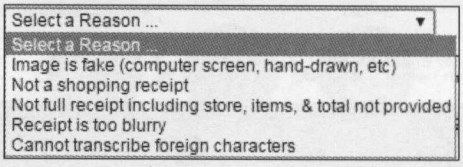
小可：嗯，用肉眼来识别这上面的内容还是比较容易的。

```
Extract purchased items from a shopping receipt
Hit Reward: $0.09 for first 20 items + Bonus: $0.01 for every 4 items.
```

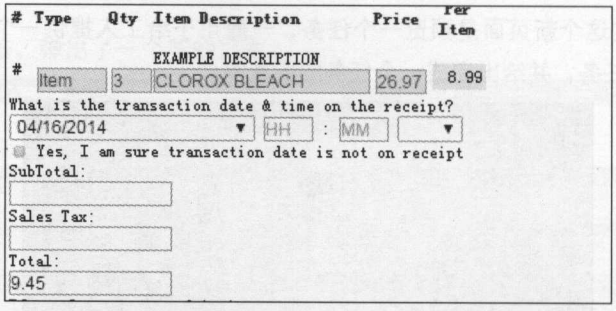
Mr. 王：看，这里任务的发布者也使用了我们前面提到过的奖励思想。每标注 20 个项目就会给回报给工人 0.09 美元的薪水，然后每完成 4 个项目就给予 0.01 美元的额外奖励。多看一些任务我们就能发现，发布者非常善于利用一些小的奖励机制来吸引工人更加认真和大量地完成的任务。

```
View example receipt to ensure you earn reward for this HIT.
Real readable original receipt   Not a receipt or not readable
```

这里任务的发布者设计还是非常严密的。有时由于图片数据的选择不当，会导致一些非收据的图片混进来，或者即使所有的图片都是收据，有些收据也会因为时间和保存不当的原因，人的肉眼难以识别。所以在任务之前，任务的发布者设计了这个问题，以免内容无法识别或者不是收据而带来不必要的麻烦。



这里也给出了无法继续进行商品标注的原因，让工人选择，也便于任务的发布者进行分类。我们在进行众包任务设计时，也要将可能出现的各种情况都考虑到，这样就会为我们后期的整理减轻很多负担。

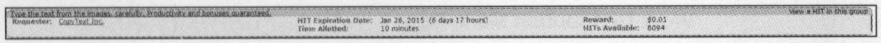


如果这是一个可以顺利识别的购物小票，工人就可以按照要求，对其中包含的商品进行标注。首先要标注小票中出现的那些商品，如类别、数量、商品描述、总价格等，其中有些部分会通过工人填进去的内容自动计算出来，比如通过商品的总价和数量计算商品的单价等。这个任务还要求工人来识别日期、总价、税额等。

小可：嗯，任务的问题设计和交互界面还是非常友好易懂的，我觉得一般的普通用户看懂

并完成这些任务还是非常容易的，只是会比较消耗时间，还真需要工人有耐心啊。

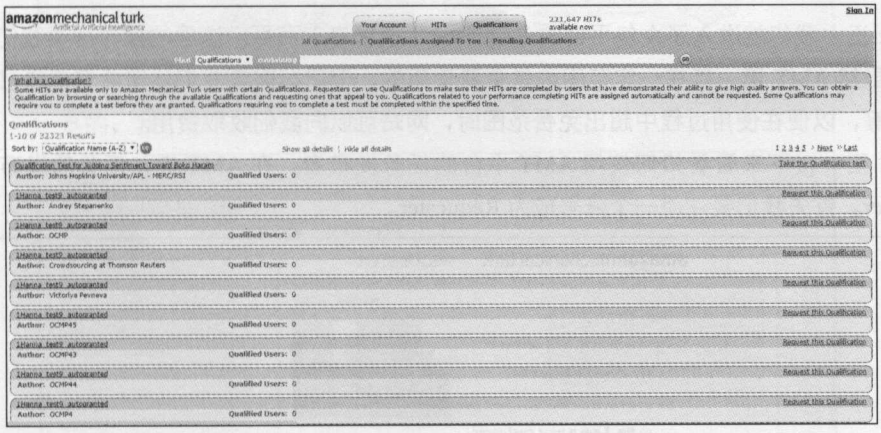
Mr. 王：毕竟是有报酬的，任务完成起来还是有一定的难度的。如果对这个任务比较感兴趣，对其报酬和工作难度都可以接受的话，点击下面的 Accept HIT 按钮就可以了。当然，为了能够顺利地进行身份认证和报酬收付，是需要进行登录的。



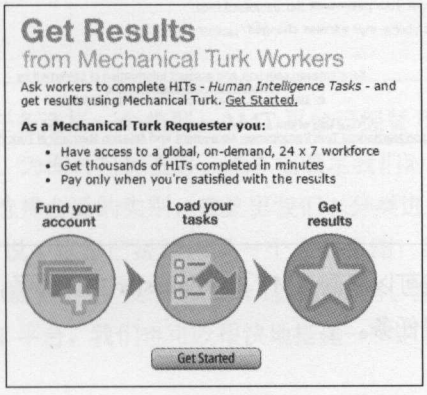
小可：再来看一个任务。咦，为什么这个任务无法查看呢？

Mr. 王：记得我们前面提到过，很多任务对精度和质量的要求是相对较高的，所以就需要用户具有一定的能力或者很强的责任心。为了验证这一点，很多任务的发布者会设置 Qualification，也就是资格认证，只有通过资格认证的人才能参与到任务的完成之中，获得相应的报酬。

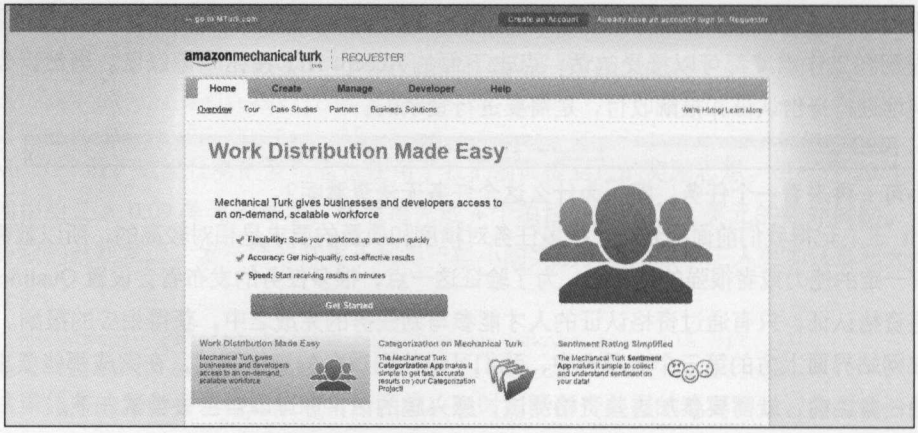
在网站界面上方的第三个选项卡中，我们可以找到很多的资格测试，在完成那些要求相对较高的任务之前，就需要参加这些资格测试，感兴趣的话，你可以自己去尝试一下。



接下来我们看看如何来发布任务。回到首页，在右侧可以看到发布任务的流程。首先要建立一个自己的账户，然后将任务加载到网站上，等到工人完成这些任务时，我们就可以得到相应的结果了。

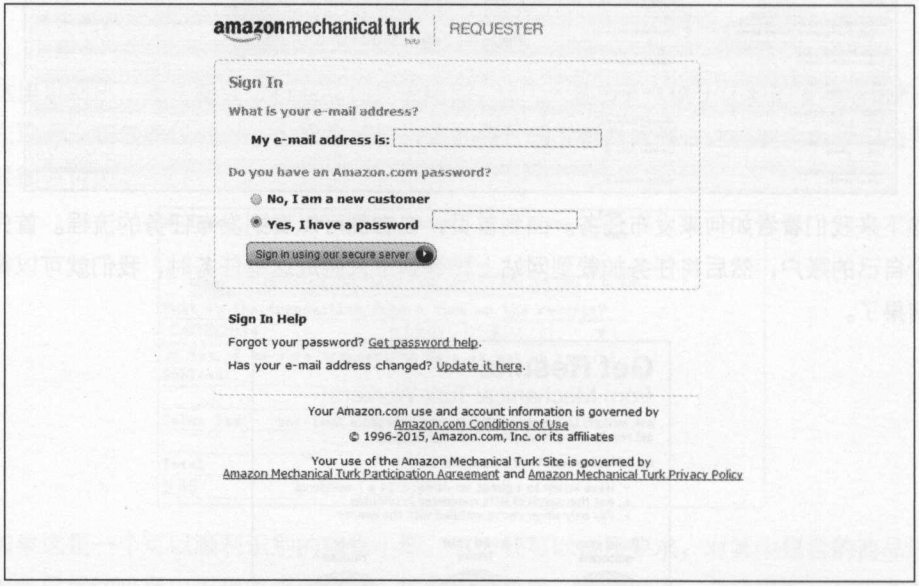


小可：点击 Get Started 按钮试试。

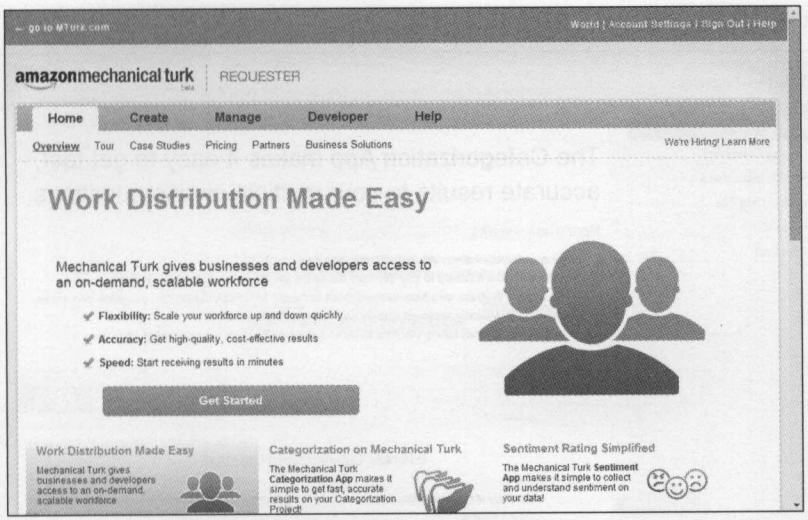


Mr. 王：我们就进入了众包平台的 Requester（请求者）页面。这里的操作需要我们先登录账户。虽然 AMT 第一年在一定范围内是免费使用的，但仍然需要我们注册并登记自己的信用卡信息等，以便在使用过程中超出免费范围时，网站可以向我们收取费用。

Mr. 王：接下来看看如何使用 AMT 平台的任务发布者。在 AMT 中，任务的发布者也叫 Requester。首先我们要注册一个账号成为 Requester。

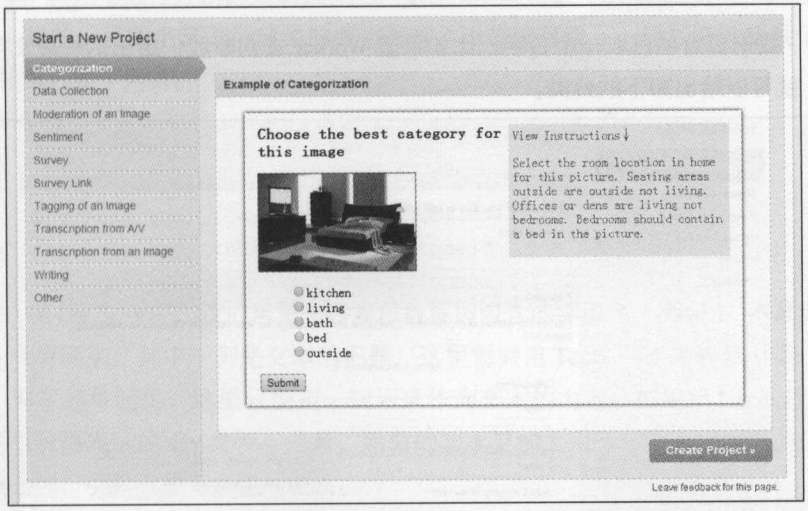


Mr. 王：登录之后我们就可以看到 AMT 的 Requester 主页面了，在这里我们可以发布任务，并寻找 Worker 来替我们完成任务。



小可：通过“Create”就可以创建新的任务了吧?

Mr. 王：是的，我们进入 Create 页面。

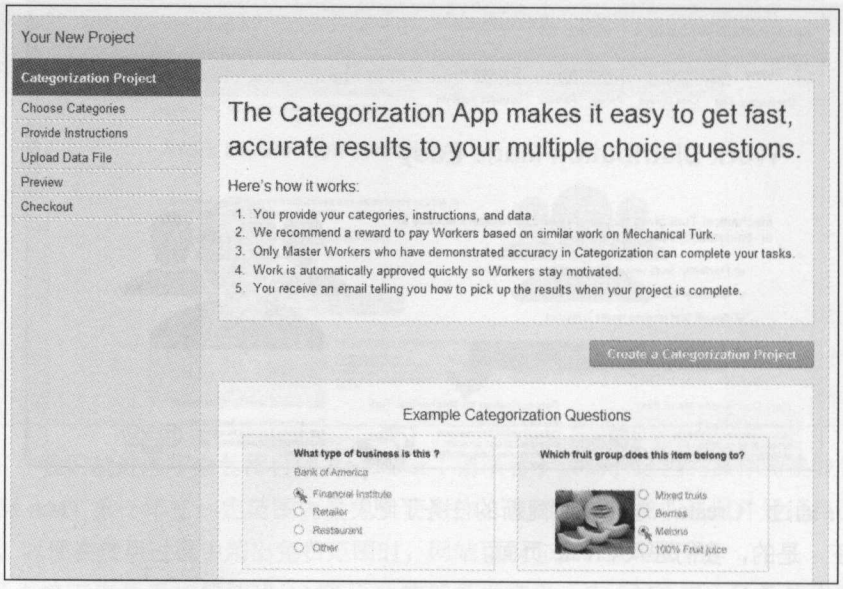


Mr. 王：首先我们要为任务选择一个类型。AMT 平台已经替我们准备了很多的任务类型，比如分类、数据收集、调研、为图片打标签等，基本能满足我们对各种众包任务的需求。可以根据自己需要完成的任务来选择合适的类别，在这里我们以分类进行举例。

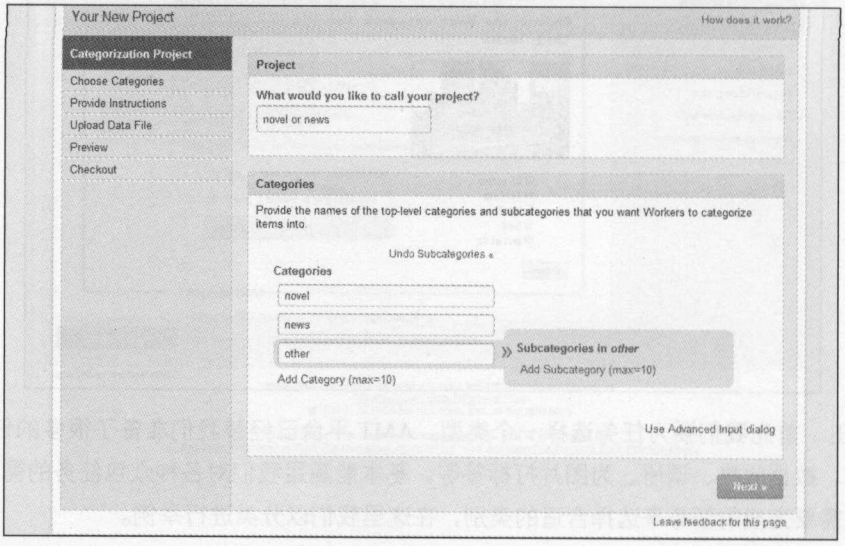
小可单击了一下分类，说：这里已经给出了一个人效果图！给出了一张图片，并且提示 Worker 给这张图片选择一个最佳分类，要求 Worker 选择是厨房、客厅、浴室、卧室、外面之一。

Mr. 王：没错，通过 AMT 平台，我们也可以很快地搭建一个这样的页面，以完成众包任务。

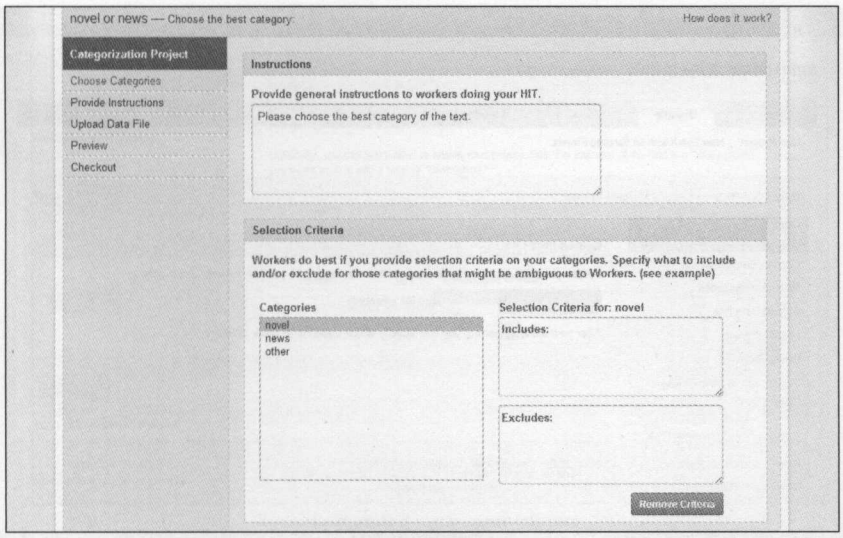
现在我们点击“Create Project”按钮。



Mr. 王：我们要为项目取一个名字，比如希望 Worker 去判断我们给出的一段文字来自小说还是新闻，就可以给出如下的分类。



Mr. 王：然后为项目给出一些提示，比如 Please choose the best category of the text.（请选择文本最合适的分类）。



Mr. 王：接下来是一个关键的步骤，我们要上传需要分类的数据文件，文件的类型是 .csv。

小可：csv 就是用逗号分割的数据文件吧？使用 Excel 也是可以打开的。

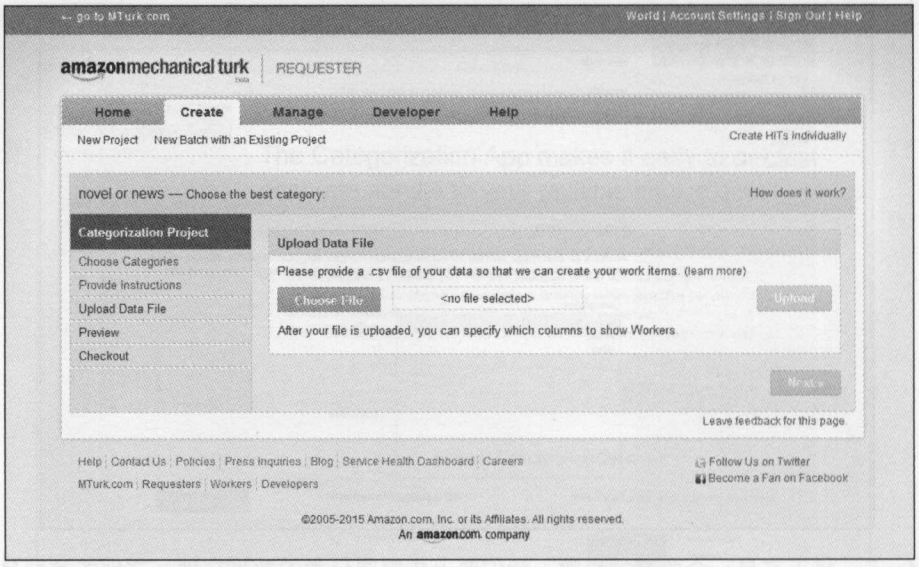
Mr. 王：是的，只要把要进行分类的数据都输入进去就可以了。比如在这里我们创建一个 csv 文件，并填入如下内容。

```
name,content
Text1,"Long long ago, there lived a king."
Text2,"The Olympic Games will be held in China."
Text3,"Long long ago, there lived a queen."
Text4,"Long long ago, there lived a prince."
Text5,"Long long ago, there lived a princess."
```

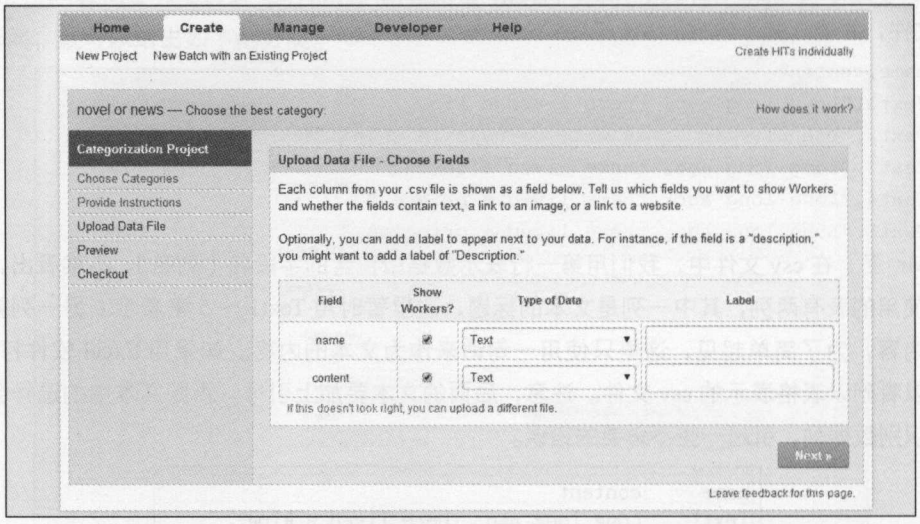
Mr. 王：在 csv 文件中，我们用第一行表示数据所包含的字段名（列名）。不难看出，我们举例使用的表有两列，其中一列是文本的标题，这里暂时用 Text1 ～ 5 来表示；另一列就是文本的内容，为了简单起见，这里只使用一句话来作为文本的内容。如果用 Excel 软件打开它，则可以看到以表格表示的 csv 文件。注意，后面的文本要加上引号，否则文本中的逗号会干扰系统识别数据列，引起一些不必要的错误。

name	content
Text1	Long long ago, there lived a king.
Text2	The Olympic Games will be held in China.
Text3	Long long ago, there lived a queen.
Text4	Long long ago, there lived a prince.
Text5	Long long ago, there lived a princess.

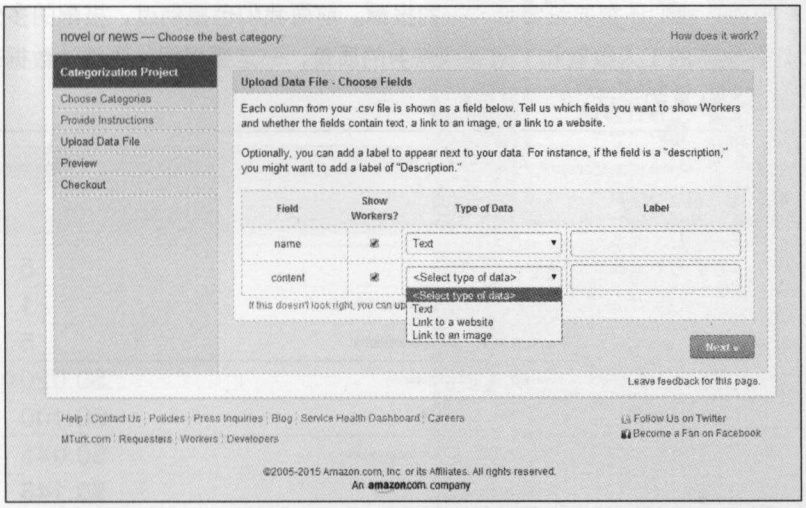
Mr. 王：我们将这个文件上传至 AMT 平台上。



Mr. 王：在文件上传完成之后，出现一个针对每一个字段的表格，其中 Show Workers 表明这个字段是否要展示给用户；Type of Data 是一个很重要的选项，我们打开看看。

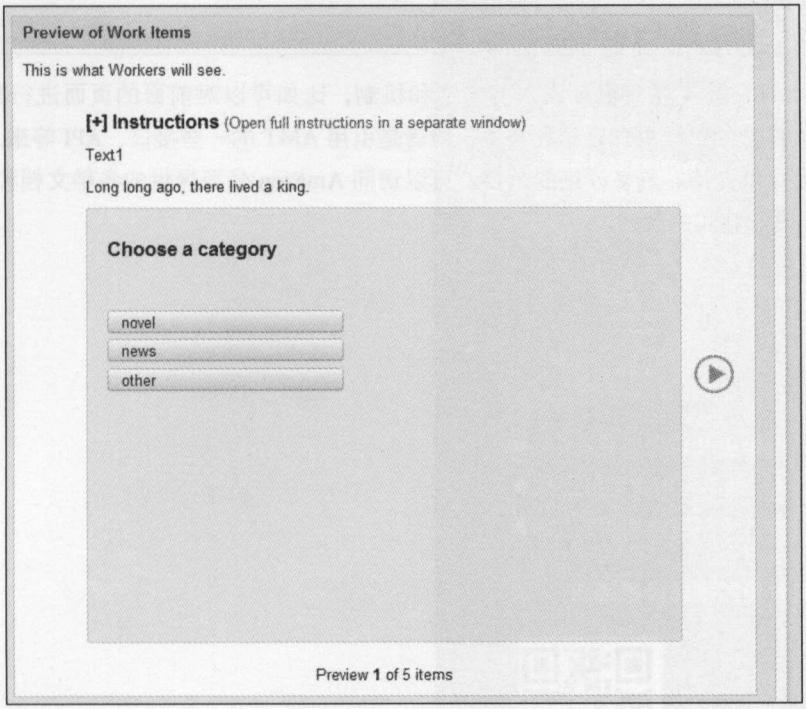


Mr. 王：可以看到，其中包括 3 个选项，即 Text、Link to a website 和 Link to an image。这里选择 Text 就好，因为我们需要工人分类的内容就是纯文本。如果需要分类的内容有图片的话，在这里就将对应的字段设为 Link to an image。同时，在 csv 文件中，我们要将对应的字段设为该图片所在的 URL 地址。

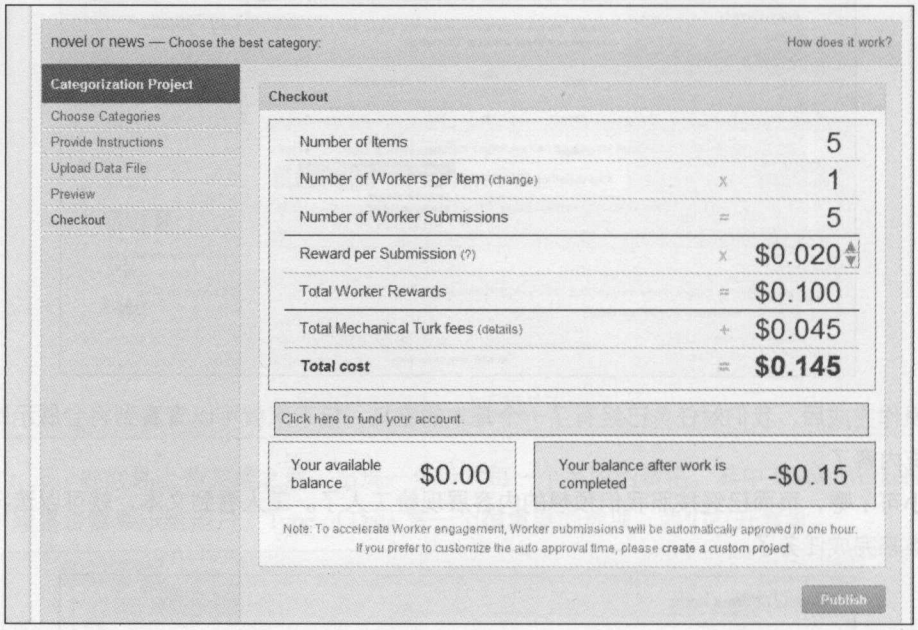


操作完成后，我们对任务已经有了一个基本的定义，接下来就可以查看到将会展示给工人的预览内容了。

小可：嗯，页面已经按照我们预想的内容展现给工人了。工人看到文本，就可以选择相应的分类来完成任务了。

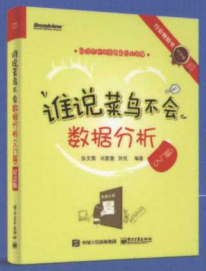


Mr. 王：最后，我们还需要设置给工人的报酬。前面我们也提到过，报酬的多少会影响到参与到完成任务之中的工人数量和工人完成任务的质量，一定要根据一些统计数据或者同类任务的报酬情况进行认真的设计。



关于 AMT，其实还有很多深入的内容和机制，比如可以对前面的页面进行美化，使得 Worker 完成任务的过程更加轻松和舒适，或者是引用 AMT 的一些接口、API 等来为我们设计的其他应用提供支持。更多详细的内容，可以访问 Amazon 公司提供的各种文档和帮助文件，里面有很丰富和翔实的解答。

我们关注职场提升的完美曲线，帮助您不断走向辉煌。

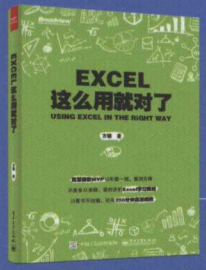


ISBN 978-7-121-28798-5
定价：59.00元

ISBN 978-7-121-28799-2
定价：59.00元

ISBN 978-7-121-23406-4
定价：59.90元

ISBN 978-7-121-25945-6
定价：66.00元



ISBN 978-7-121-27441-1
定价：69.90元

◀《小强升职记（升级版）：时间管理故事书（双色）》
◀《Excel图表拒绝平庸（纪念版）》
◀《思维导图的三招十八式》

联系系列故事的策划人，只需随手一拍：

邮箱：info@dozan.cn
新浪微博：@长颈鹿27
支持网站（知了帮）：zhiliaobang.com

Broadview®
www.broadview.com.cn
博文视点·IT出版旗舰品牌
技术凝聚实力·专业创新出版

作者娓娓道来，揭开大数据的七彩面纱。背景、计算复杂性、内外磁盘算法、MapReduce并行计算、Spark　平台、众包、大数据挖掘……精心选择的主题，涵盖大数据算法的主要方面。内容有浅有深，讲述引人入胜。从零开始学大数据算法不仅可能，而且非常有趣。

美国北德克萨斯大学教授 黄艳

本书堪称大数据领域的一本教科书，作者深入、系统地讲述了大数据领域的算法体系，从概念剖析、算法理论、应用实例到工程实践。此书一改传统算法书籍平铺直叙的笔法风格，用师生对话的新颖形式，形成了很强的“代入感”，让读者身临其境地体验算法学习的乐趣。本书不但是大数据算法初学者的启蒙教材，而且是大数据工程师高价值的参考读本。

中国电信北京研究院 灯塔大数据中心主任 孙静博

“算法未动，数据先行”。数据作为不可或缺的资源，在实际应用中占据了越来越重要的位置。尤其是大数据，近年来，诸多研究领域已经验证了其蕴藏着意想不到的金矿。作者在大数据处理领域有超过十年的研究经历，本书有对大数据中复杂问题的精辟理解，从中可以看出深厚的功底。作者将大数据研究领域中的诸多模糊、深奥的概念和原理通过自然聊天的方式，由浅入深娓娓道来。因此，本书非常适合作为读者快速了解或学习大数据的入门首选。

百度资深工程师 刘占一

零基础，但有广度。大数据处理的方方面面，从图灵机原理到常用数据结构及算法，从经典并行处理框架到时兴的众包计算，本书都有精彩阐述。有深度，但不枯燥。生动鲜活的对话体，处处充满形象的比喻和实例，在愉悦中轻松进阶大数据处理的理论、应用和实践。

微软亚洲研究院 主管研究员 邵斌

博文视点Broadview
新浪微博 weibo.com
@博文视点Broadview

策划编辑：张月萍
责任编辑：葛　娜
封面设计：吴海燕

上架建议：计算机>算法

ISBN 978-7-121-28937-8
9 787121 289378 >
定价：59.00元